

USENIX

conference

.....
proceedings

Proceedings of the 2008 USENIX Annual Technical Conference

2008 USENIX Annual Technical Conference

Boston, MA, USA

June 22–27, 2008

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Boston, MA, USA June 22–27, 2008

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is \$45 for members and \$55 for nonmembers.
Outside the U.S.A. and Canada, please add \$20 per copy for postage (via air printed matter).

Thanks to Our Sponsors

SILVER SPONSORS



BRONZE SPONSORS



SPONSORS



IBM Research

Thanks to Our Media Sponsors

ACM Queue
Addison-Wesley Professional/
Prentice Hall Professional/
Cisco Press
BetaNews
Dr. Dobb's Journal
Free Software Magazine
Homeland Defense Journal

IEEE Security & Privacy
InfoSec News
ITToolbox
Linux+DVD Magazine
Linux Gazette
Linux Journal
Linux Pro Magazine
LXer.com

Messaging News
No Starch Press
SNIA
SourceForge
StorageNetworking.org
The Register
UserFriendly.org

© 2008 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN-13: 978-1-931971-59-1

USENIX Association

**Proceedings of the
2008 USENIX Annual Technical Conference
(USENIX '08)**

**June 22–27, 2008
Boston, MA, USA**

Conference Organizers

Program Co-Chairs

Rebecca Isaacs, *Microsoft Research*
Yuanyuan Zhou, *University of Illinois at Urbana-Champaign*

Program Committee

Frank Bellosa, *University of Karlsruhe*
Jeff Chase, *Duke University*
Dawson Engler, *Stanford University*
Jason Flinn, *University of Michigan*
Keir Fraser, *University of Cambridge*
Steve Gribble, *University of Washington*
Liviu Iftode, *Rutgers University*
Arkady Kanevsky, *NetApp*
Angelos Keromytis, *Columbia University*
Emre Kiciman, *Microsoft Research*
Sam King, *University of Illinois at Urbana-Champaign*
Jeff Mogul, *HP Labs*
Erich Nahum, *IBM T.J. Watson Research Center*
David Presotto, *Google*

Sean Rhea, *Meraki, Inc.*
Erik Riedel, *Seagate Research*
Timothy Roscoe, *ETH Zürich*
Mike Swift, *University of Wisconsin*
John Wilkes, *HP Labs*
Emmett Witchel, *University of Texas*
Xiaolan (Catherine) Zhang, *IBM T.J. Watson Research Center*
Zheng Zhang, *Microsoft Research*

Poster Session Chairs

Emre Kiciman, *Microsoft Research*
Sam King, *University of Illinois at Urbana-Champaign*

Training Program

Dan Klein, *USENIX*

Invited Talks

Dan Klein and Ellie Young, *USENIX*

The USENIX Association Staff

External Reviewers

David Bacon
Paul Barham
Muli Ben-Yehuda
Aditya Bhandari
Sumeer Bhola
Ricardo Bianchini
Aniruddha Bohra
Cristian Borcea
Nedyalko Borisov
Mike Burrows
Shiva Chaitanya
Pau-Chen Cheng
Mihai Christodorescu
Ionut Constandache
Anthony Cozzie
Fred Douglass

Dan Ellard
Sameh Elnikety
Jeffrey Eрман
Douglas Freimuth
Vinod Ganapathy
Chris Grier
Chuanxiong Guo
Owen S. Hofmann
Simon Kellner
Terence Kelly
Gerd Lieflaender
Zorik Machulsky
Prince Mahajan
Varun Marupadi
Andreas Merkel
Raphael Neider

Thu Nguyen
Donald E. Porter
Hany E. Ramadan
Christopher J. Rossbach
Yaoping Ruan
Jiri Schindler
Shubho Sen
Charles Shen
Asia Slowinska
Keith Smith
Mike Spreitzer
Lex Stein
Christopher Stewart
Marc Stiegler
Jan Stoess
Tom Talpey

Shuo Tang
Radu Teodorescu
Vamsidhar Thummala
Bhuvan Urganekar
Wietse Venema
Geoff Voelker
Kaladhar Voriganti
Charles P. Wright
Yongqiang Xiong
Danfeng Yao
Ben-Ami Yassourl
Jian Yin
Lintao Zhang

2008 USENIX Annual Technical Conference

June 22–27, 2008

Boston, MA, USA

Index of Authors	vi
Message from the Program Co-Chairs	vii

Wednesday, June 25

Virtualization

Decoupling Dynamic Program Analysis from Execution in Virtual Environments	1
<i>Jim Chow, Tal Garfinkel, and Peter M. Chen, VMware</i>	
Protection Strategies for Direct Access to Virtualized I/O Devices	15
<i>Paul Willmann, Scott Rixner, and Alan L. Cox, Rice University</i>	
Bridging the Gap between Software and Hardware Techniques for I/O Virtualization	29
<i>Jose Renato Santos, Yoshio Turner, and G. (John) Janakiraman, HP Labs; Ian Pratt, University of Cambridge</i>	

Disk Storage

Idle Read After Write—IRAW	43
<i>Alma Riska and Erik Riedel, Seagate Research</i>	
Design Tradeoffs for SSD Performance	57
<i>Nitin Agrawal, University of Wisconsin—Madison; Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy, Microsoft Research, Silicon Valley</i>	
Context-Aware Mechanisms for Reducing Interactive Delays of Energy Management in Disks	71
<i>Igor Crk and Chris Gniady, University of Arizona</i>	

Network

Optimizing TCP Receive Performance	85
<i>Aravind Menon and Willy Zwaenepoel, EPFL</i>	
ConfIDNS: Leveraging Scale and History to Detect Compromise	99
<i>Lindsey Poole and Vivek S. Pai, Princeton University</i>	
Large-scale Virtualization in the Emulab Network Testbed	113
<i>Mike Hibler, Robert Ricci, Leigh Stoller, and Jonathon Duerig, University of Utah; Shashi Guruprasad, Cisco Systems; Tim Stack, VMware; Kirk Webb, Morgan Stanley; Jay Lepreau, University of Utah</i>	

File and Storage Systems

FlexVol: Flexible, Efficient File Volume Virtualization in WAFL	129
<i>John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas, NetApp, Inc.</i>	
Fast, Inexpensive Content-Addressed Storage in Foundation	143
<i>Sean Rhea, Meraki, Inc.; Russ Cox and Alex Pesterev, MIT CSAIL</i>	
Adaptive File Transfers for Diverse Environments	157
<i>Himabindu Pucha, Carnegie Mellon University; Michael Kaminsky, Intel Research Pittsburgh; David G. Andersen, Carnegie Mellon University; Michael A. Kozuch, Intel Research Pittsburgh</i>	

Thursday, June 26

Web and Internet Services

Handling Flash Crowds from Your Garage171
Jeremy Elson and Jon Howell, Microsoft Research

Remote Profiling of Resource Constraints of Web Servers Using Mini-Flash Crowds185
Pratap Ramamurthy, University of Wisconsin—Madison; Vyas Sekar, Carnegie Mellon University; Aditya Akella, University of Wisconsin—Madison; Balachander Krishnamurthy, AT&T Labs—Research; Anees Shaikh, IBM Research

A Dollar from 15 Cents: Cross-Platform Management for Internet Services199
Christopher Stewart, University of Rochester; Terence Kelly and Alex Zhang, Hewlett-Packard Labs; Kai Shen, University of Rochester

Workloads and Benchmarks

Measurement and Analysis of Large-Scale Network File System Workloads213
Andrew W. Leung, University of California, Santa Cruz; Shankar Pasupathy and Garth Goodson, NetApp Inc.; Ethan L. Miller, University of California, Santa Cruz

Evaluating Distributed Systems: Does Background Traffic Matter?227
Kashi Venkatesh Vishwanath and Amin Vahdat, University of California, San Diego

Cutting Corners: Workbench Automation for Server Benchmarking241
Piyush Shivam, Sun Microsystems; Varun Marupadi, Jeff Chase, Thileepan Subramaniam, and Shivrath Babu, Duke University

Short Papers

Power-aware Remote Replication for Enterprise-level Disaster Recovery Systems255
Kazuo Goda and Masaru Kitsuregawa, The University of Tokyo

A Linux Implementation Validation of Track-Aligned Extents and Track-Aligned RAID5261
Jin Qian, Christopher Meyers, and An-I Andy Wang, Florida State University

Automatic Optimization of Parallel Dataflow Programs267
Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava, Yahoo! Research

A TCP-layer Name Service for TCP Ports275
Sérgio Freire, PT Inovação/IEETA/University of Aveiro; André Zúquete, IEETA/IT/University of Aveiro

Using Causality to Diagnose Configuration Bugs281
Mona Attariyan and Jason Flinn, University of Michigan

Diverse Replication for Single-Machine Byzantine-Fault Tolerance287
Byung-Gon Chun, ICSI; Petros Maniatis, Intel Research Berkeley; Scott Shenker, University of California, Berkeley

Friday, June 27

Security and Bugs

Vx32: Lightweight User-level Sandboxing on the x86293
Bryan Ford and Russ Cox, Massachusetts Institute of Technology

LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages307
Yan Tang, Qi Gao, and Feng Qin, The Ohio State University

Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing321
Dan Wendlandt, David G. Andersen, and Adrian Perrig, Carnegie Mellon University

Spectator: Detection and Containment of JavaScript Worms335
Benjamin Livshits and Weidong Cui, Microsoft Research

Memory and Buffer Management

A Compacting Real-Time Memory Management System349
Silviu S. Craciunas, Christoph M. Kirsch, Hannes Payer, Ana Sokolova, Horst Stadler, and Robert Staudinger, University of Salzburg

Prefetching with Adaptive Cache Culling for Striped Disk Arrays363
Sung Hoon Baek and Kyu Ho Park, Korea Advanced Institute of Science and Technology

Context-Aware Prefetching at the Storage Server377
Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza, University of Toronto

Wide-Area Systems

Free Factories: Unified Infrastructure for Data Intensive Web Services391
Alexander Wait Zaranek, Tom Clegg, Ward Vandeweye, and George M. Church, Harvard University

Wide-Scale Data Stream Management405
Dionysios Logothetis and Kenneth Yocum, University of California, San Diego

Experiences with Client-based Speculative Remote Display419
John R. Lange and Peter A. Dinda, Northwestern University; Samuel Rossoff, University of Victoria

Index of Authors

Agrawal, Nitin	57	Kaminsky, Michael	157	Riska, Alma	43
Akella, Aditya	185	Kanevsky, Arkady	129	Rixner, Scott	15
Amza, Cristiana	377	Kelly, Terence	199	Rossoff, Samuel	419
Andersen, David G.	157, 321	Kirsch, Christoph M.	349	Santos, Jose Renato	29
Attariyan, Mona	281	Kitsuregawa, Masaru	255	Sekar, Vyas	185
Babu, Shivnath	241	Kozuch, Michael A.	157	Shaikh, Anees	185
Baek, Sung Hoon	363	Krishnamurthy, Balachander	185	Shen, Kai	199
Chase, Jeff	241	Lange, John R.	419	Shenker, Scott	287
Chen, Peter M.	1	Lentini, James	129	Shivam, Piyush	241
Chow, Jim	1	Lepreau, Jay	113	Silberstein, Adam	267
Chun, Byung-Gon	287	Leung, Andrew W.	213	Smith, Keith A.	129
Church, George M.	391	Livshits, Benjamin	335	Sokolova, Ana	349
Clegg, Tom	391	Logothetis, Dionysios	405	Soundararajan, Gokul	377
Cox, Alan L.	15	Manasse, Mark	57	Srivastava, Utkarsh	267
Cox, Russ	143, 293	Maniatis, Petros	287	Stack, Tim	113
Craciunas, Silviu S.	349	Marupadi, Varun	241	Stadler, Horst	349
Crk, Igor	71	Menon, Aravind	85	Staudinger, Robert	349
Cui, Weidong	335	Meyers, Christopher	261	Stewart, Christopher	199
Davis, John D.	57	Mihailescu, Madalin	377	Stoller, Leigh	113
Dinda, Peter A.	419	Miller, Ethan L.	213	Subramaniam, Thileepan	241
Duerig, Jonathon	113	Olston, Christopher	267	Tang, Yan	307
Edwards, John K.	129	Pai, Vivek S.	99	Turner, Yoshio	29
Ellard, Daniel	129	Panigrahy, Rina	57	Vahdat, Amin	227
Elson, Jeremy	171	Park, Kyu Ho	363	Vandewege, Ward	391
Everhart, Craig	129	Pasupathy, Shankar	213	Vishwanath, Kashi Venkatesh	227
Fair, Robert	129	Payer, Hannes	349	Wang, An-I Andy	261
Flinn, Jason	281	Perrig, Adrian	321	Webb, Kirk	113
Ford, Bryan	293	Pesterev, Alex	143	Wendlandt, Dan	321
Freire, Sérgio	275	Poole, Lindsey	99	Willmann, Paul	15
Gao, Qi	307	Prabhakaran, Vijayan	57	Wobber, Ted	57
Garfinkel, Tal	1	Prakash, Ashish	129	Yocum, Kenneth	405
Gniady, Chris	71	Pratt, Ian	29	Zaraneek, Alexander Wait	391
Goda, Kazuo	255	Pucha, Himabindu	157	Zayas, Edward	129
Goodson, Garth	213	Qian, Jin	261	Zhang, Alex	199
Guruprasad, Shashi	113	Qin, Feng	307	Zúquete, André	275
Hamilton, Eric	129	Ramamurthy, Pratap	185	Zwaenepoel, Willy	85
Hibler, Mike	113	Reed, Benjamin	267		
Howell, Jon	171	Rhea, Sean	143		
Janakiraman, G. (John)	29	Ricci, Robert	113		
Kahn, Andy	129	Riedel, Erik	43		

Message from the Program Co-Chairs

Welcome to the 2008 USENIX Annual Technical Conference in Boston! ATC continues to be a venue where researchers and system builders submit their top work in computer systems, resulting in an exciting technical program with a practical flavor.

This year the program contains 34 excellent papers—28 regular and 6 short papers—selected from a total of 176 submissions. This is around 30% more than previous years and conforms with the trend of an increasing number of worthy papers being submitted to premier system conferences. The topics range from memory and buffer management to handling flash crowds from your garage. The program also features two invited speeches: a keynote from David Patterson about the current parallel revolution, and a plenary closing session by Matthew Melis about the Columbia accident investigation and returning NASA's spaces shuttle to flight.

As always, the success of this conference is dependent on the guidance and help provided by the outstanding USENIX staff, who have been a great pleasure to work with. We were also blessed with a hard-working and thorough program committee of 24 leading researchers in the field, including 11 representatives from various industrial companies and labs. Due to the unexpectedly large number of submissions, each PC member reviewed about 33 papers, for a total of 782 reviews. Similarly to other recent conferences, we divided the submission process into two stages. The first stage assigned three reviews for each paper; based on these reviews, we rejected 49 papers. The second stage assigned two more reviews for each of the remaining 127 papers. Although each PC member was responsible for all their assignments, we did solicit about 70 reviews from external experts. Among all 34 accepted papers, only four are PC-authored papers.

We thank USENIX staff members, PC members, and reviewers for their hard work under time pressure during this process. Their contributions are very valuable, not only to authors by providing useful feedback, but also to the community by building a high-quality program. Special thanks are due to Eddie Kohler for the paper submission and reviewing system, and to Shan Lu and Joseph Tucek for hosting it at UIUC. In addition, Ellie Young has provided essential guidance throughout and, with Dan Klein, has also assembled a set of great invited talks. We also thank Sam King and Emre Kiciman for organizing the poster session.

Finally, we would like to express our appreciation to our industry sponsors for their help in making this event possible and enjoyable. In particular we thank Google, Microsoft Research, HP Invent, and WhiteOak Technologies, Inc., for their generous support.

Rebecca Isaacs, Microsoft Research
Yuanyuan Zhou, University of Illinois at Urbana-Champaign
USENIX '08 Co-Chairs

Decoupling dynamic program analysis from execution in virtual environments

Jim Chow Tal Garfinkel Peter M. Chen
VMware

Abstract

Analyzing the behavior of running programs has a wide variety of compelling applications, from intrusion detection and prevention to bug discovery. Unfortunately, the high runtime overheads imposed by complex analysis techniques makes their deployment impractical in most settings. We present a virtual machine based architecture called *Aftersight* ameliorates this, providing a flexible and practical way to run heavyweight analyses on production workloads.

Aftersight decouples analysis from normal execution by logging nondeterministic VM inputs and replaying them on a separate analysis platform. VM output can be gated on the results of an analysis for intrusion prevention or analysis can run at its own pace for intrusion detection and best effort prevention. Logs can also be stored for later analysis offline for bug finding or forensics, allowing analyses that would otherwise be unusable to be applied ubiquitously. In all cases, multiple analyses can be run in parallel, added on demand, and are guaranteed not to interfere with the running workload.

We present our experience implementing *Aftersight* as part of the VMware virtual machine platform and using it to develop a realtime intrusion detection and prevention system, as well as an offline system for bug detection, which we used to detect numerous novel and serious bugs in VMware ESX Server, Linux, and Windows applications.

1 Introduction

Dynamic program instrumentation and analysis enables many applications including intrusion detection and prevention [18], bug discovery [11, 26, 24] and profiling [10, 22]. Unfortunately, because these analyses are executed inline with program execution, they can substantially impact system performance, greatly reducing their utility. For example, analyses commonly used for detecting buffer overflows or use of undefined memory routinely incur overheads on the order of 10-

40x [18, 26], rendering many production workloads unusable. In non-production settings, such as program development or quality assurance, this overhead may dissuade use in longer, more realistic tests. Further, the performance perturbations introduced by these analyses can lead to Heisenberg effects, where the phenomena under observation is changed or lost due to the measurement itself [25].

We describe a system called *Aftersight* that overcomes these limitations via an approach called *decoupled analysis*. Decoupled analysis moves analysis off the computer that is executing the main workload by separating execution and analysis into two tasks: *recording*, where system execution is recorded in full with minimal interference, and *analysis*, where the log of the execution is replayed and analyzed.

Aftersight is able to record program execution efficiently using virtual machine recording and replay [4, 9, 35]. This technique makes it possible to precisely reconstruct the entire sequence of instructions executed by a virtual machine, while adding only a few percent overhead to the original run [9, 35]. Further, as recording is done at the virtual machine monitor (VMM) level, *Aftersight* can be used to analyze arbitrary applications and operating systems, without any additional support from operating systems, applications, compilers, etc.

Aftersight supports three usage models: synchronous safety, best-effort safety, and offline analysis. First, for situations where timely analysis results are critical (e.g., intrusion detection and prevention), *Aftersight* executes the analysis in parallel with the workload, with the output of the workload synchronized with the analysis. This provides synchronous safety that is equivalent to running the analysis inline with the workload. Second, for situations that can tolerate some lag between the analysis and the workload, *Aftersight* runs the analysis in parallel with the workload, with no synchronization between the output of the workload and the analysis. This best-effort safety allows the workload to run without be-

ing slowed by the analysis. Often analyses whose performance impact would be prohibitive if done inline can run with surprisingly minimal lag if run in parallel. Third, Aftersight can run analyses offline for situations where analyses are not known beforehand or are not time critical, such as when debugging.

Aftersight is a general-purpose analysis framework. Any analysis that can run in the critical path of execution can run in Aftersight, as long as that analysis does not change the execution (this would break the determinism that Aftersight's replay mechanism relies upon). Also, Aftersight makes the entire system state at each instruction boundary available for analyses, providing greater generality than approaches based on sampling. Further, logs originating from the VMM can be replayed and analyzed in different execution environments (e.g., a simulator or VMM). This flexibility greatly eases program instrumentation and enables a variety of optimizations.

We have implemented an Aftersight prototype on the x86 architecture, building on the record and replay capability of VMware Workstation. Our framework enables replay on the QEMU whole-system emulator, which supports easy instrumentation during replay and analysis. With this framework, we have implemented an online security analysis that can be used to detect buffer overflow attacks on running systems. We also implemented an analysis that can perform checks for memory safety and heap overflows, and we used this analysis to discover several new and serious bugs in VMware ESX Server, Linux, and Windows applications.

2 The case for decoupled analysis

Aftersight improves dynamic analysis by decoupling the analysis from the main workload, while still providing the analysis with the identical, complete sequence of states from the main workload. This combination of decoupling and reproducibility improves dynamic analysis in the following ways.

First, Aftersight allows analyses to be added to a running system without fear of breaking the main workload. Because Aftersight runs analyses on a separate virtual machine from the main workload, new analyses can be added without changing the running application, operating system, or virtual machine monitor of the main workload.

Second, Aftersight offers users several choices along the safety/performance spectrum. Users who can tolerate some lag between the analysis and the workload can improve the performance of the workload and still get best-effort safety or offline analysis, while users who require synchronous safety can synchronize the output of the workload with the analysis.

Third, with best-effort safety or offline analysis, Aftersight can improve latency for the main workload by mov-

ing the work of analysis off the critical path. Because analyses no longer slow the primary system's responsiveness, heavyweight analyses can now be run on realistic workloads and production systems without fear of perturbing or unduly slowing down those workloads. For example, system administrators can use intensive checks for data consistency, taint propagation, and virus scanning on their production systems. Developers can run intensive analyses for memory safety and invariant checking as part of their normal debugging, or as additional offline checks that augment testing that must already be performed in a quality-assurance department. As an extreme illustration of the type of heavyweight analysis enabled by Aftersight, computer architects can capture the execution of a production system with little overhead, then analyze the captured instruction stream on a timing-accurate, circuit-level simulator. Even when providing synchronous safety, Aftersight can sometimes improve performance compared to running the analysis inline by leveraging the parallel execution of the workload and the analysis.

Fourth, Aftersight increases the parallelism available in the system by providing new ways to use spare cores. Aftersight can run an analysis in parallel with the main workload, and it can run multiple analyses in parallel with each other.

Fifth, Aftersight makes it feasible to run multiple analyses for the exact same workload. Without Aftersight, the typical way to run multiple analyses is to conduct a separate run per analysis, but this suffers from the likelihood of divergent runs and inconsistent analyses. Aftersight, in contrast, guarantees that all analyses operate on the same execution. In addition, each analysis takes place independently, so programmers need not worry about unforeseen interactions between the analyses. Nor must they worry about perturbing the source workload with their analysis. Aftersight allows the number of simultaneous analyses to scale with the number of spare processors in a system, all while not affecting the performance of the primary system.

Sixth, Aftersight makes it possible to conduct an analysis that was not foreseen during the original run. This *ex post facto* style of analysis is particularly powerful when it is difficult to anticipate exactly what must be analyzed. For example, analyzing computer intrusions invariably requires one to examine in detail a scenario that was not foreseen (else, one would have prevented the intrusion). Debugging performance or configuration problems leads to a similar need for conducting unforeseen analysis. Aftersight allows the user to iteratively develop and run new analyses, all on the same exact execution.

Finally, by decoupling analysis from the main execution, Aftersight allows the analysis and execution components to be individually optimized to their in-

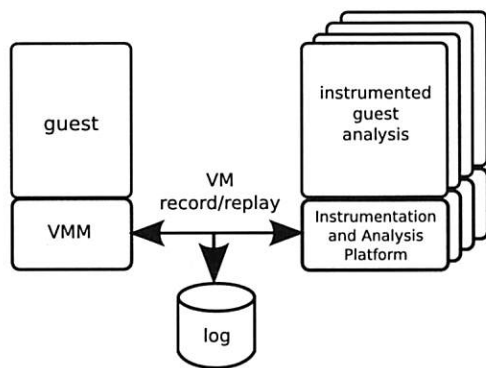


Figure 1: Overview of current system.

tended function. The main workload execution can be performed on a platform optimized for realtime performance and responsiveness (such as a commercial VMM), while analysis can be delegated to a platform optimized for ease of instrumentation (such as an extensible simulator).

3 Architectural overview

Our current Aftersight system targets the *x86* architecture and has three main components: the virtual machine monitor (VMM), deterministic VM record/replay, and an analysis and instrumentation framework. For our prototype, each of these pieces builds on functionality of existing off-the-shelf components. In this section we examine aspects of these components that are relevant for decoupled analysis. In Section 5 we look at how they can be modified and integrated to facilitate decoupled analysis.

3.1 VMM

A VMM provides an environment for running arbitrary guest operating systems and applications in a software abstraction of the hardware [23]. No modifications to a standard VMM are required to support decoupled analysis, except for the need to support replay. Aftersight uses VMware Workstation, a highly optimized production *x86* VMM. VMware Workstation uses a hosted architecture [28], i.e. it uses a *host* operating system to access physical devices like disk or network.

As Aftersight is based on a virtual machine platform, it inherits a variety of useful and desirable traits. First, any individual process in the guest VM as well as the guest OS kernel itself can be a target of Aftersight instrumentation and analysis. Furthermore, a range of target systems are supported without extra work: a single implementation works across OSes, processes, languages, etc.

Next, virtualization is becoming increasingly ubiquitous in a wide range of computing environments. Because Aftersight can be provided as a service of the virtual infrastructure with nominal overhead, there is a rel-

atively easy path to adoption. With such a primitive in place, the deployment of new monitoring and analysis tools can be a continuous, normal part of the execution of guests.

Finally, operating at the VMM level gives Aftersight visibility at all layers of the software stack. It can be used to analyze operating systems, applications, and interactions across components. This generality is critical for applications ranging from performance analysis, to tracking the flow of sensitive or potentially malicious data in a system.

3.2 Deterministic VM record and replay

Aftersight builds upon the replay facilities in VMware Workstation¹. A deterministic VM record/replay system records enough information about a running workload to reproduce its exact instruction sequence. To support replay, a VM must record and replay all inputs to the CPU that are not included in the state of the guest memory, registers, or disk. This includes reads from external devices, such as the network, keyboard, or timer, and asynchronous events such as interrupts. Recording these nondeterministic inputs enable VM replay to recreate the whole instruction stream [4, 9, 35]. As with other software-based replay systems [9], VMware Workstation is not able to replay virtual multiprocessors.

VM replay systems are highly efficient in time and space overhead [9, 35]. This efficiency comes about because nearly all instructions produce the same result given the same inputs, and most instructions use only the results of previous instructions as their inputs. Because of this domino effect, a long sequence of instructions can be exactly reproduced while only supplying a few values that come “from outside” the system.

A study [35] of VMware’s then-current replay implementation showed performance overheads for SPEC benchmarks as low as 0.7% and an average of 5% [35]. Another replay implementation for the *x86* [9] reported similar overheads. Overheads will generally be workload dependent, however. Worst-case performance observed in [35] reached 31% and 2.6x for some workloads. However, many of the chief bottlenecks were not fundamental [35], and subsequent improvements have lowered these overheads.

Trading the overhead of analysis for the overhead of VM replay is a compelling exchange for many heavyweight analyses. However, even for lightweight analyses, the ability to run multiple or *ex post facto* analyses still provides reason to use decoupled analysis.

While Aftersight uses VM replay, replay can also be implemented at many levels besides the VM-level, such as the OS process-level [27], the JVM-level [8], or the

¹First released in VMware Workstation 6, where it was an experimental feature.

disk level [32]. VM replay stands out for a couple reasons.

First, a single VM replay implementation enables one to replay the entire state of all software on that hardware platform regardless of operating system, language or run-time environment. In contrast, other types of replay can only work for a small subset of available software as they are heavily dependent on the particular language and operating system they are designed for. Reimplementing replay for each OS or language variant would be a herculean task.

Second, a VM replay system often has lower overhead than higher-level recording. For example, one could replay the file system at the system-call level, but this would require all file system calls to be recorded, including every `read` of every file. However, a VM-based system need only record nondeterministic VM inputs, and this frees it from recording reads from the file cache or disk. Similarly, a process-level record/replay system must record reads from IPC pipes, files, and all system calls that return data, while these can all be ignored by a VM-based solution.

3.3 Analysis framework

Aftersight does instrumentation and analysis dynamically during replayed execution. Normally in a VM record/replay system, the same VMM is used during both recording and replaying. A key property of Aftersight is its ability to support *heterogenous replay*, i.e. the ability to use one platform to execute and record a workload and a different platform to replay and analyze, with each platform tuned for its particular purpose.

The Aftersight prototype relies on a VMM for execution and recording, whereas replay and analysis can be done in a VMM or a simulator. A VMM is an excellent platform for recording, because it is optimized to minimize recording overhead to support production environments. However, platforms such as software simulators are often better suited to supporting general-purpose analysis.

For example, many VMM environments don't provide a simple, low overhead way to instrument every memory access. This can be implemented on top of page protections and faulting on every memory access. However, a software simulator can often accomplish this task faster and with less effort than a VMM will natively.

Analysis environments Dynamic instrumentation can be implemented in many ways. Most simply, we can build ad-hoc hooks into our replaying environment that supply callbacks when events of interest happen.

In our Aftersight prototype, we implement dynamic instrumentation through dynamic binary translation (BT). BT is the technique of dynamically translating a set of instructions into an alternate set of instructions

on-the-fly, which are then executed. Techniques such as caching translations [33] can be used to make this process very efficient. Affecting what translations are produced allows one to very flexibly instrument a running program.

Our prototype offers two BT environments to analysis applications: one is based on VMware Workstation, the other on QEMU [3], which is an open-source x86 simulator. Both offer the ability to run code using BT alone, or in some combination with native execution [1]. However, each has its own strengths and limitations.

VMware's BT is extremely fast, but it is optimized for performance rather than extensibility. For example, VMware's BT does not support an extensible intermediate representation (IR). An extensible IR is commonly used in general purpose BT systems [17, 3] to abstract the x86's CISC-style instructions into a more instrumentation-friendly RISC-style format. However, these additional translation costs make little sense given VMware's specialized use of BT. Also, for efficiency reasons, VMware BT runs in ring 0, and in an environment where dynamic memory allocation is heavily constrained. Developing general-purpose analyses under these constraints is quite burdensome, and the resulting analyses may even be slower from having to work with limited memory.

In contrast, QEMU is not nearly as fast as VMware Workstation, but it is much more flexible: it provides an extensible IR and runs as a regular user-mode process, which means normal program facilities like `malloc`, `gdb`, etc. are available. In converting it to enable replay, we stripped out much of its now unnecessary functionality, to the point where it is little more than a simple CPU simulator. The virtual device model, including the disk, network, the chipset, and the local APIC, have all been removed. All that remains are the components needed to deal with instruction execution and memory access.

Of course, because analysis is decoupled, other special-purpose analysis environments could be built to better suit the needs of particular analyses if desired.

4 Online analysis

In two of Aftersight's three usage models, synchronous safety and best-effort safety, the analysis runs in parallel with the workload. Aftersight makes it easy to simultaneously record and analyze a workload. In our prototype, recording generates a replay log on disk. Analysis VMs can run on separate cores and process the log as it is being generated by the primary VM. Analysis can even take place across multiple machines by reading the log file over the network, since network bandwidth is more than adequate for most workloads. Log sizes are often quite modest [9, 35]. For example, Xu et al. [35] notes that only 776 KB of compressed log space was nec-

essary to record an entire Windows XP bootup-shutdown sequence.

This section describes how Aftersight synchronizes the main workload with the analysis when the two are running in parallel, and how running the analysis in parallel with the workload can speed up the analysis.

4.1 Synchronization

When running in simultaneous record and analysis mode, analysis results may affect the operation of the primary VM (e.g., a security check may detect an intrusion and halt the system). When this feedback is needed, Aftersight can take one of two strategies to synchronize the execution of the primary and analysis VMs.

The need for synchronization arises because the primary VM executes ahead of the analysis VM. The portion of the primary VM's execution that has not yet been run on the analysis VM is *speculative*. This speculative portion will usually be committed by the analysis VM as checks complete. In the rare case when checks fail, the speculative portion of execution differs from what would have been executed with inline analysis.

The first method for synchronizing provides synchronous safety, which is equivalent to running the analysis inline with the workload. To provide this guarantee, Aftersight defers the output of the primary VM (e.g., network packets) while they are speculative, i.e., until the analysis reaches the same point in execution. Deferring outputs while they are speculative ensures that the released outputs of the primary VM are identical to those of a system with inline checks, even though the internal state of the primary VM may differ from a system with inline checks [19].

In addition to synchronizing the primary's output with the analysis VM, we could also limit how far the primary is allowed to run ahead of the analysis VM. Limiting the lag between primary and backup limits the amount of time that the primary's outputs are deferred, which in turn limits the amount of timing perturbation the primary VM may observe (e.g., when it measures the round-trip time of a network).

Deferring output in the above manner provides the same safety guarantee as if the analysis were running inline with the workload. However, it may hurt performance by blocking the output of the primary VM.

A different point in the safety/performance spectrum optimizes performance but relaxes the safety guarantee by giving lazy feedback to the primary VM. In this case, the main workload executes at full speed and is not slowed by the work of analysis. Rather, analysis results are fed back to the main workload as they become available. This usage model is useful when the analysis is too heavyweight to run with stronger safety guarantees, or when the analysis does not require such guarantees, as in

profiling or debugging.

4.2 Accelerating analysis

The analysis VM in Aftersight executes the same instructions as the primary VM, and it also does the work of analysis. Because the analysis VM is doing more work than the primary VM, it can easily become a bottleneck, especially when providing synchronous safety. This section describes several ways to improve the performance of the analysis VM, to allow it to better keep up with the primary VM.

First, a surprising amount of performance can be won from a basic aspect of replayed VM execution: interrupt delivery is immediate. x86 operating systems use the `hlt` instruction to wait for interrupts; this saves power compared to idle spinning. During analysis, `hlt` time passes instantaneously. One `hlt` invocation waiting for a 10ms timer interrupt can consume equal time to tens of millions of instructions on modern 1+GHz processors. Section 6.3 provides more detail on the boost this can have on performance.

Second, device I/O can be accelerated during replay. For example, network writes need not be sent, and network reads can use data from the replay log. This frees guests from waiting for network round-trip times, especially because disk throughput is often greater than end-to-end network throughput. Disk reads can similarly be satisfied from the replay log rather than from the analysis VM's disk, and this can accelerate the analysis VM because the replay log is always read sequentially. This optimization can also free the analysis VM from executing disk writes during replay, which frees up physical disk bandwidth and allows write completion interrupts to be delivered as soon as the guest arrives at an appropriate spot to receive them. Disk reads done by the primary VM may also prefetch data and thereby accelerate subsequent reads by the analysis VM [5].

Third, a number of opportunities allow Aftersight to *memoize* operations that happen during record that don't need to be fully replayed. An example of this is exception checking.

There are many times where the x86 needs to check for exceptional conditions. Although these checks rarely raise exceptions, executing them adds considerable overhead in our CPU simulator. Segment limit checks are an example: every memory reference or instruction fetch must be checked that it is within bounds for an appropriate segment ².

²These add enough overhead that QEMU completely ignores the behavior. This turns out to work for many workloads, but not all. Playing fast and loose with the specification in this way inevitably causes failures—non-executable stacks are popularly implemented with segment limits in major x86 Unix derivatives where non-executable page protections are unavailable (which is true for all non-PAE kernels), but QEMU makes them behave incorrectly.

Decoupled analysis allows one to reduce the overhead of exception checking on the analysis VM by leveraging the exception checking that has already occurred on the main VM. The time and location in the instruction stream of any exceptions are recorded by the main VM, and these exceptions are delivered during replay just like asynchronous replay events. This strategy frees the analysis VM from the overhead of explicitly checking for exceptions during replay. Memoizing these checks makes the CPU simulator faster and less complex, while still guaranteeing proper replay of a workload that contains violations of the checks.

There are many x86 checks that can be memoized, although we have not yet implemented this optimization in Aftersight: debug exceptions, control transfer checks for segment changes, the alignment check (which when enabled, ensures all memory accesses are performed through pointers aligned to appropriate boundaries), and others.

5 Implementation and integration

While Aftersight builds on existing components, leveraging these for decoupled analysis poses a variety of challenges. This section discusses how to adapt a simulation environment to replay VMM logs and the challenges posed by the heterogeneous combination of record and replay components.

Aftersight uses different platforms for recording and analysis. For recording, Aftersight uses VMware Workstation, which is designed to minimize the time and space overhead of recording. For analysis, Aftersight can use VMware Workstation or QEMU. Simple analyses can be conducted by modifying VMware Workstation's BT, while more general analyses are easiest to implement in QEMU, which is designed for flexibility rather than pure speed.

For replay and analysis, compatibility is an issue for both platforms. When VMware Workstation replays a log, no compatibility issues arise with devices, chipset, etc. because the emulation code is identical. However, because it relies directly on the hardware for CPU emulation, replay is generally infeasible if the processor is significantly different (e.g., attempting to replay a log from an Intel CPU on an AMD platform). In contrast, with a CPU simulator like QEMU we can easily support a wide range of CPU families on a single hardware platform. However, QEMU does not have the same device models as the recording platform. In this next section, we look at how Aftersight bridges the compatibility gap between the VMware Workstation recording and QEMU in two areas: I/O device emulation and hardware performance counters.

5.1 Device emulation

The first gap between our recording platform (VMware Workstation) and one of our analysis platforms (QEMU) is device emulation. QEMU emulates different I/O devices than VMware, which prevents QEMU from directly consuming the log recorded by VMware.

To understand the problem and the solution we adopted, it is helpful to consider two different methods for recording device interactions. The first method is to record all outputs from an emulated device to the CPU. During replay, the recorded values would be resupplied to the CPU (presumably the guest OS device drivers). This method is ideal for compatibility between the recording and analysis platform because no device emulation is needed during replay.

However, VMware Workstation and other VM replay systems [9] use a second method to record and replay device interactions. Instead of recording the output from the emulated devices, they record the nondeterministic, external inputs to those devices. During replay, these recorded inputs are redelivered to the devices, and these allow the emulated devices to be deterministically replayed along with the CPU.

VM replay systems use this second method for two reasons. A main reason is that the second method allows a replaying session to "go live"—to stop replaying and start responding to new input—at any point while replaying. In contrast, recording and replaying the outputs of the emulated device without replaying the emulated device itself means that the emulated device is not available to go live. Another reason is that it can drastically reduce the amount of data that must be recorded. For example, to replay a disk read operation, the first method must record the actual data being read from the emulated disk, while the second method need only record the nondeterministic inputs to the disk (note that the inputs from the CPU to the disk are deterministic and need not be recorded).

Unfortunately, recording only the nondeterministic inputs to the device leads to a compatibility problem during analysis. Whereas the VM recording system assumes that the replaying system can replay the emulated device, QEMU and other flexible analysis systems usually will not emulate the exact same devices used during recording.

Aftersight bridges the compatibility gap between recording and analysis for devices by adding a *relogging* step to replay. We modified VMware's replay system to record a new log during replay, which contains all outputs from the emulated device to the CPU, including responses to I/O requests, interrupt delivery, and effects on memory. This log is equivalent to one generated by the first method of replaying devices and has the same compatibility advantages, i.e. the analysis system needs no

device emulation during replay.

While our modified VMware VMM supports relogging, none of our modifications to support relogging impact the *record* side operation of the VMM, since relogging is only active during replay.

5.2 Hardware performance counters

The second gap between our recording platform (VMware Workstation) and one of our analysis platforms (QEMU) is hardware performance counters. VM replay implementations will normally use hardware counters to determine when a nondeterministic event happens during recording, as well as to trigger that event during replay [4, 9]. These counters record aspects of the dynamic instruction stream that help to uniquely position an event in time such as the instruction count [4], or the number of branches executed [9].

Instructions added dynamically by BT, as well as by analysis instrumentation, disrupt counts kept by the hardware by adding dynamic instructions in an unpredictable manner. This makes hardware counters difficult to use directly by our CPU simulator.

Instead, our CPU simulator emulates the accounting provided by the hardware counters in the translations it emits. These translations include a small amount of code to update counts and dispatch to an event handler when it is time to deliver an asynchronous nondeterministic event (such as an interrupt or DMA).

QEMU doesn't normally allow interrupt delivery within a basic block [2] of instructions. Instead, these events are delayed until the current basic block completes. The VMware recording system contains no such artificial restriction, so we needed to remove this restriction of QEMU to replay VMware's log. When our stripped-down QEMU reaches a basic block containing a replay event, it will emit new translations for the block. The block is split into two halves: the block of instructions before the replay event, and the block after. Checks between basic blocks will determine that the BT system can deliver the event.

6 Evaluation

Aftersight makes it possible to run heavyweight analyses on realistic workloads with several options along the safety/performance spectrum. In this section, we evaluate the performance of Aftersight under three usage models. We first show how Aftersight can provide synchronous safety with slightly higher performance than a system using inline analysis. Next, we show how Aftersight with best-effort safety makes it possible to run heavyweight analyses in parallel with the main workload and how the techniques described in Section 4.2 allow heavyweight analyses to keep up with the main workload. Last, we demonstrate the utility of enabling heavy-

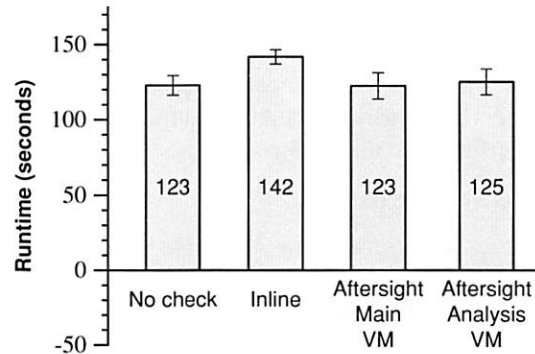


Figure 2: Aftersight performance with synchronous safety.

weight analyses that are built and applied after the main workload completes.

6.1 Synchronous safety

We first evaluate how Aftersight performs when providing safety that is equivalent to running the analysis inline with the main workload. In this usage model, Aftersight runs the analysis in parallel with the main workload, and defers the output of the workload until the analysis reaches that output. Aftersight's main benefits for this usage model are the ability to add new analyses without fear of breaking the workload and the ability to conduct later analyses that were not envisioned at the time of the run.

When providing synchronous safety, Aftersight's performance is limited by the analysis VM. A reasonable expectation is that the performance of the analysis VM will be comparable to that of an inline system (or slower, due to replaying overhead). While this can be true for many workloads, the analysis VM in Aftersight can also run faster than an inline system by taking advantage of the work done by the primary VM (Section 4.2). We demonstrate an example of this phenomenon through the following experiment.

We evaluate Aftersight with synchronous safety on a workload that uses `wget` to fetch a directory of linked web pages from a local `lighttpd` web server. The directory of web pages consists of 5000 HTML files, each 200 KB. The workload starts with a cold file cache and spends most of its time fetching data from disk. The check running in the analysis VM mimics a trivial on-access virus scanner by computing for 2 ms on each disk request.

Figure 2 compares the performance for Aftersight with synchronous safety with running the analysis inline with the workload. The analysis VM in Aftersight always trails the primary VM that it is replaying, so the workload is considered complete when the analysis VM completes.

Although the analysis VM is slowed by the overhead of replaying, it leverages the disk reads performed by the primary VM to regain this performance. The net effect on this benchmark is that Aftersight achieves slightly better performance than inline analysis.

6.2 Best-effort safety

We next demonstrate how Aftersight enables heavy-weight analyses to execute concurrently with a workload with best-effort safety. Our analysis enforces protection for guest address spaces at the granularity of individual bytes of memory. This supports checking for a wide range of memory errors, though we only apply it to catching heap overflows in our example.

An in-memory bitmap specifies whether each byte of a particular address space is writable or not. The bitmap is organized as a two-level page table to conserve space. To implement the checks, the analysis dynamically instruments instructions that write to memory. These writes are translated to look up the appropriate protection bits in the bitmap and check if they allow writing. If they do, the write proceeds normally, otherwise the analysis invokes an error handler. When running the analysis in parallel with the main workload (online), the error handler can invoke a feedback action that takes corrective measures. For example, the system can automatically suspend the primary VM when an error is detected.

We implemented the analysis in VMware Workstation by modifying the binary translation done during replay. On each write, the translation saves two scratch registers and the CPU flags, checks the protection bits in a bitmap, then restores the two scratch registers.

To use this analysis, the guest kernel specifies the desired protection for each byte of memory of an address space. We modified a guest Linux kernel to use this facility to do heap-overflow bounds checks on dynamically allocated kernel objects. Linux already includes a facility for adding “red zone” buffers to the start and end of every `kmalloc` object. We use this facility and add code to set the bitmap permission bits for these redzones appropriately. Only kernel code needs checking, since normal page protections prevent user code from touching kernel heap objects.

Without Aftersight, this analysis is too slow to be used in production settings. We measured the speed of analysis without Aftersight by running the checks inline in the VMware binary translator of the main workload. The results are shown in Figure 3. For a kernel compilation benchmark, running the benchmark with the analysis inline (i.e. without Aftersight) takes 191.10 seconds (a 2.48x slowdown compared to running the benchmark without the analysis). Running the workload with Aftersight reduces the time to complete the benchmark to 84.86 seconds. With best-effort safety, Aftersight re-

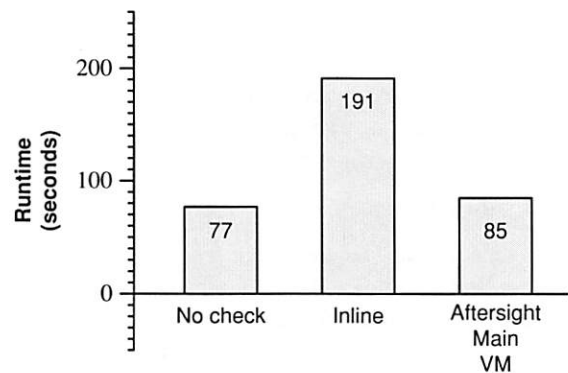


Figure 3: Comparison of kernel compile overhead with heap buffer overflow detection: Aftersight runs a kernel compilation benchmark nearly 2x faster than inline checking. Checks are still run concurrently with variable lag.

duces the perturbation on the main workload by moving the overhead of analysis off the critical path of the main workload and into a separate analysis system. The main workload pays for this in the form of record/replay overhead, which for this benchmark is 10.4%.

A potential disadvantage of decoupling this analysis from the main workload is that the detection of a write violation may occur long after the offending instruction execution. However, even delayed feedback can be very useful. For example, we ran an SSH server in the primary VM, logged into it from an outside client, and invoked a system call that contains an erroneous heap overflow. As before, we check byte-level write protections in an analysis VM. We measured how long it takes the analysis VM to discover the problem, assuming that prior to the SSH connection, both the primary and analysis VMs are synchronized to the same point in time.

In Figure 4 we see the progress of the primary and analysis VM, measured in #branches executed vs. wall clock time. As shown by the horizontal distance on the graph between identical branch counts in the primary and analysis VM, the analysis VM lags the primary VM by varying amounts during the run. In this experiment, there was a 0.86 second latency between the write violation in the primary VM and its detection in the analysis VM.

Decoupled analysis can lead to delays between an event in the main VM and the analysis of those events in the analysis VM. This delay is not a problem for analyses that don’t need to provide feedback to the main VM, or for analyses that have no response time requirement on their feedback (such as optimization or bug finding). The delay is not ideal for analyses that implement security or correctness checks. However, delayed feedback can still be very useful, and is certainly preferable to being unable

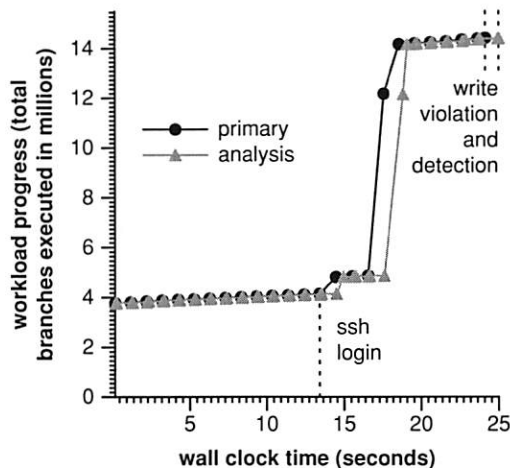


Figure 4: Measuring latency of simultaneous record and analysis for write protection violations: The two dotted lines at the end show when the write violation occurs and when it is identified by the analysis (shown separated by 0.86 seconds).

to run heavyweight analyses at all.

6.3 Idle-time boost

Section 6.2 showed that decoupling analysis from the main workload with best-effort safety can improve the performance of the main workload, but that the analysis may complete significantly after the workload. Since heavyweight analyses may be much slower than the main workload, a natural question is “How can heavyweight analyses keep up with the main workload over a long run?”. Section 4.2 describes some techniques that help a replaying analysis VM to keep up with a main VM, and this section evaluates the effectiveness of these techniques. In particular, we observe that idle time in the primary can provide enough time for the analysis VM to catch up during replay, even for very heavyweight analyses.

As an example of a heavyweight analysis platform, we configure the VMware Workstation binary translator to translate *all* guest instructions (instead of just guest kernel instructions), but still with a relatively small (2.9 MB) code cache. Even with no extra analysis, this configuration of VMware Workstation runs several times slower than the normal configuration.

We measured the overhead of this analysis system on a CPU-bound workload, which is winLAME (a GUI front-end to the LAME mp3-encoder) in a Windows XP VM, encoding .wav files into mp3 format. The test machine is a two-processor, dual core 2GHz AMD Opteron (containing four logical CPUs) running Debian Linux. On this workload, the analysis VM takes 4.65x as long to

complete the workload as the main VM, due to the overhead of the slower binary translator,

Most realistic workloads can be replayed much faster by skipping over idle time in the main workload. To see how idle time can provide a boost, we ran this analysis again with an interactive, desktop workload. In the primary VM, we use Windows XP and:

- Start Firefox. Edit the proxy settings to get out of the corporate network.
- Visit slashdot.org, scroll through the front page, browsing for one minute.
- Visit internal website and download an Excel spreadsheet containing numbers used in this paper.
- Close Firefox. Start Excel, open the spreadsheet.
- Create a chart, and plot two curves using data in the spreadsheet. Add a trend line to one of the curves.
- Close Excel. Open Powerpoint, and create a custom animation using four block arrows flying in from different directions. Close Powerpoint.

Figure 5 shows the results of this experiment. Figure 5(a) shows the progress of the primary VM and the analysis VM as wall clock time progresses. Horizontal gaps between the two curves occur where bursts of high CPU utilization cause analysis to lag the primary.

Figure 5(b) more clearly illustrates these bursts by showing the instantaneous compute rates of the primary and the analysis. Figure 5(b) shows that the primary contains many compute spikes. Meanwhile, the analysis runs at a more constant pace because it is limited by the speed of binary translation. These compute spikes can cause significant lag (one spike causes the main VM to execute 6x faster than the analysis VM). However, as shown in Figure 5(a), the idle times in the workload allows the analysis VM to eventual catch up from this lag.

Figure 5(c) graphs the lag between the main VM and the analysis VM as the workload progresses. This graph demonstrates two major benefits of decoupled analysis with best-effort safety. First, note that the analysis VM can lag behind the main VM significantly (10-11 seconds on average, and as high as 35 seconds behind). These periods of high lag imply that running the analysis inline with the main VM or with synchronous safety would cripple the interactive performance of the main VM (imagine waiting 35 seconds between clicking a button and waiting for the corresponding menu to appear!). In contrast, Aftersight with best-effort safety decouples the analysis, so that users of the primary VM don’t experience this lag. Instead, the primary’s responsiveness is completely independent of the speed of analysis, e.g. button clicks and menu selections occur at full speed.

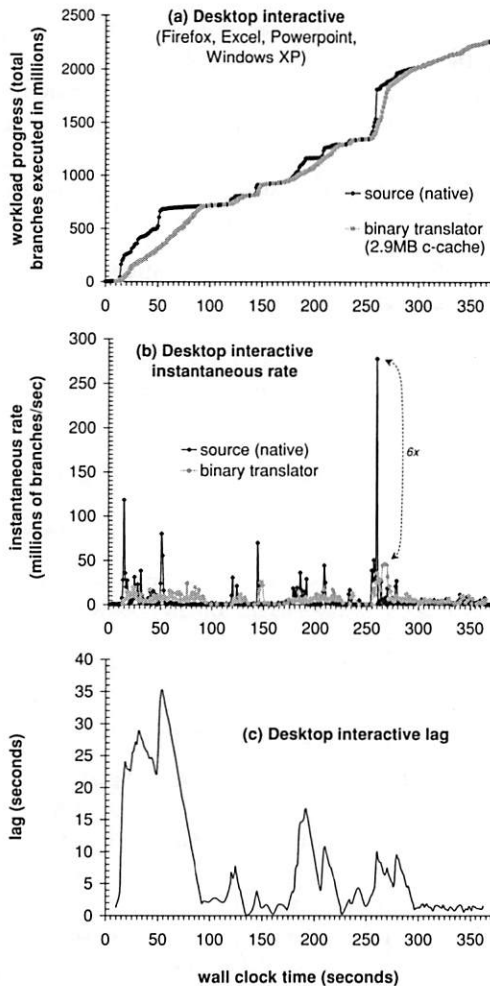


Figure 5: Plotting the progress of a source workload, and a online tandem analysis of that workload running on another core: a desktop interactive session shows how analysis (running in a binary translator that is several times slower) is allowed to lag during bursty periods of computing, and catch up when bursts end.

Second, this graph illustrates how idle times between bursts of CPU activity enable the analysis VM to catch up with the main VM. Although the lag gets as long as 35 seconds during this workload, the analysis VM is able to catch up by the end of the workload. Catching up in this manner is only possible because of decoupled analysis. Synchronizing the main VM with the analysis VM would limit the speed of the main VM during bursts of CPU activity to that of the analysis VM, and it would limit the speed of the analysis VM during idle periods to that of the (idling) main VM.

Idle time in real-world workloads is quite common. In an informal poll, idle time was over 95% for several desktop computers used by full-time computer program-

mers for compiling programs, editing tex, e-mail, web browsing, and running VMs. Idle time was over 75% for a production web server and mail server at the EECS department of a large public university.

Idle time can also be deliberately increased in many systems, and this may help heavyweight analyses keep up with the main VM. For example, idle time can be increased in server farms by adding more servers and balancing load across them.

6.4 Offline analysis

This section demonstrates how Aftersight enables heavyweight analyses to be built and applied after the main workload completes.

To illustrate this capability, we implemented an analysis that ensures every instruction executed by the source workload meets a set of memory safety guarantees: that a dereferenced pointer must point to valid stack or heap data, that any bit of stack or heap data used in control flow or as a pointer/index must be initialized before use, and that there are no memory leaks (roughly, that the guest executes “Valgrind-safe” [26]).

Asserting continual satisfaction of these constraints is a very expensive job—properly implementing a check for uninitialized data requires a full taint propagation analysis [26]. Our tool implements this taint analysis and does it at bit-level precision, largely following the implementation given by [26]. The analysis tracks the state of each bit in all guest memory and registers according to the state machine shown in Figure 6. To initialize the state of each bit, the checker interposes on all memory allocation requests for the heap (through calls to memory allocator functions) and stack (through manipulations of the stack pointer). To maintain the value of the state of each bit, the checker interposes on every instruction executed by the workload, for example to propagate the state of source memory or registers to destination memory or registers. The tool uses symbol information to identify calls to the appropriate heap allocator for the system. When analyzing a particular user process, the target process first identifies itself by making a hypercall.

The analysis is built using Aftersight’s QEMU-based CPU simulator. The analysis is extremely heavyweight (on the order of 100x) and is best suited to running offline.

Implementing this analysis in Aftersight yields two important benefits relative to traditional tools such as Valgrind and Purify. First, Valgrind and Purify slow their target program too much to be used for long-running, realistic workloads, and they may perturb their target programs too much to capture realistic interactive workloads. In contrast, Aftersight allows long-running, interactive workloads to run with little overhead by allowing the work of analysis to be run later. Second, tools such

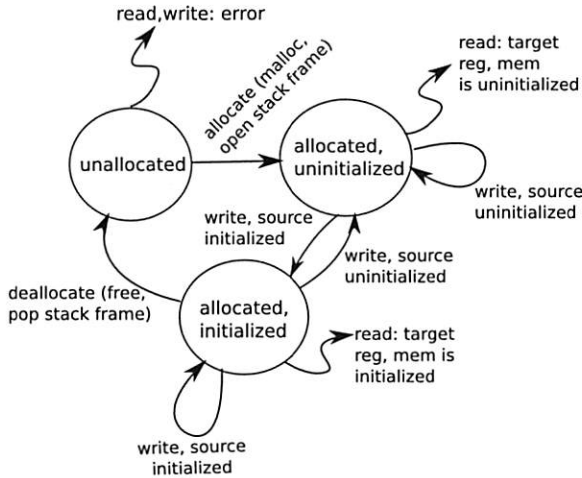


Figure 6: Memory consistency checker: a state machine is associated with every bit of guest memory and registers, and finds uses of garbage data and dangling pointers.

as Valgrind and Purify can only be applied to user-level code. They cannot be applied to OS or VMM code, even though such code is critical to reliability and safety. In contrast, analyses implemented in Aftersight can be applied to all software running in the virtual machine, including the operating system or even another VMM running inside VMware Workstation.

We have used this tool to find serious bugs in large, complex systems, including kernel code such as VMware ESX Server and Linux. The rest of this section describes bugs we found with this analysis.

ESX Server We used our analysis tool in the development of VMware ESX Server [30] by running ESX inside a VM hosted by VMware Workstation. We found 10 type safety errors, over half of which were classified as critical or show-stopper bugs, and were able to fix them during development.

For example, in one bug the ESX Server kernel has a utility data structure for recording statistics whose use is sprinkled throughout the code. It takes an array of values as argument, and stores it:

```

Histogram_New(..., const uint32 numBuckets,
               const Histogram_Datatype*
               const bucketLimits) {
    ...
    histo = Heap_Alloc(heap,
                       sizeof(struct Histogram) +
                       bucketCountsSz);
    if (histo != NULL) {
        histo->numBuckets = numBuckets;
        ...
        histo->limits.arbitrary.bucketLimits =
            bucketLimits;
    }
}

```

...

This array of values is used when manipulations to the statistics occur. Unfortunately, the structure is allocated on the heap, and some callers initialize it with an array from the stack:

```

SCSIAllocStats(Heap_ID heap,
                ScsiStats *stats) {
    Histogram_Datatype limits[...];
    ...
    stats->cmdSizeHisto =
        Histogram_New(..., limits);
    ...
}

```

Under certain circumstances, this would cause a crash, but the tool was able to diagnose the problem without reproducing the crash by noticing that using the data structure caused references to data located in popped off stack frames.

Linux We also applied our analysis tool to the Linux kernel by running it as the guest kernel in a VM. Our tool diagnosed a long-overlooked type safety error in an old part of the core Linux kernel. Its UDP stack makes use of uninitialized stack garbage on reception of UDP packets through `recvfrom`. Whenever `recvfrom` is called, a `msg` structure containing a field `msg_flags` would be allocated on the stack, but never initialized:

```

/* from net/socket.c */
asmlinkage long sys_recvfrom(...) {
    ...
    struct msghdr msg;
    ...
    msg.msg_control = NULL;
    msg.msg_controllen = 0;
    msg.msg_iovlen = 1;
    msg.msg_iov = &iiov;
    iiov.iov_len = size;
    iiov.iov_base = ubuf;
    msg.msg_name = address;
    msg.msg_namelen = MAX_SOCKET_ADDR;
    if (sock->file->f_flags & O_NONBLOCK)
        flags |= MSG_DONTWAIT;
    err = sock_recvmsg(sock, &msg, size, flags);
    ...
}

```

Following the call stack down through `sock_recvmsg`, this structure is passed to `udp_recvmsg`, which uses `msg_flags`:

```

Backtrace:
#0 udp_recvmsg
    (linux-2.6.20.1/net/ipv4/udp.c:843)
#1 sock_common_recvmsg
    (linux-2.6.20.1/net/core/socket.c:1617)
#2 sock_recvmsg
    (linux-2.6.20.1/net/socket.c:630)
#3 sys_recvfrom
    (linux-2.6.20.1/net/socket.c:1608)
#4 sys_socketcall
    (linux-2.6.20.1/net/socket.c:2007)

```

```
#5 syscall_call
(linux-2.6.20.1/arch/i386/kernel/entry.S:0)

/* net/ipv4/udp.c */
int udp_rcvmsg(..., struct msghdr *msg, ...)
{
    ...
    if (... && msg->msg_flags&MSG_TRUNC) {
    ...

```

In this case, the test on `msg_flags` gated a checksum computation, so although no crash would result from this erroneous use of `msg_flags`, it would cause random, unnecessary extra computation to occur on reception of UDP packets.

We discovered this bug in Linux 2.6.20.1 and reported it to kernel developers, who fixed it in the next major release. This bug existed in the code for many years (all prior versions of 2.6 and all versions of 2.4 we checked back to 2002).

Putty We also tested a common Windows SSH client called Putty. Our tool found one memory leak that was invoked whenever a menu item was selected. We are currently investigating other user programs as well.

7 Future work

There are many interesting open questions about how to optimize and synchronize record and analysis.

Workload memoization Memoizing state generated by native hardware during record can avoid the need for re-computation during analysis, and this can be used to accelerate analysis. We use this kind of memoization when simulating SMM (system management) mode [13]. Our CPU simulator does not implement *x86*'s SMM mode because the version of QEMU we started with did not support it. However, the VMware VMM is more faithful about this part of the architecture, and some target workloads do contain execution in SMM. Aftersight memoizes SMM to maintain compatibility for these workloads. In its relogging step, Aftersight records the changes to memory made in SMM (some of which may be used outside SMM, for example by the guest BIOS code). During replay, the analysis infrastructure reproduces these effects at the proper time, thus avoiding the need to simulate SMM code.

In addition to helping compatibility, memoization can also be used to accelerate replay. For example, consider an analysis where we are only interested in the execution of a specific user process. With memoization, we could use relogging to summarize the execution of all other code in the system into their effects on memory and registers. In essence, this turns the execution of the OS and other processes into the equivalent of a single DMA operation. This would allow subsequent replay and analyses to ignore writes to pages not mapped into the current

process, context switches to other processes, and most kernel activity. Focusing on one process would also allow us to accelerate the simulator by not emulating the hardware MMU. Instead we could simply use `mmap` or its equivalent to set up the address space for the process and allow memory accesses to run natively.

Feedback modes Simultaneous record and analysis mode can use different types of feedback loops to synchronize. We discussed the tradeoffs between two basic approaches, blocking and lazy feedback modes, in Section 4. However, one limitation of these approaches is that they fail to take into account the semantics of the OS, application, or analysis.

If we add more intelligence to our synchronization strategy we can take advantage of natural join points that occur. For example, when analyzing a web server for security, we can impose a synchronization restriction that the analysis VM must be in-synch with the primary whenever the primary initiates an *outgoing* TCP connection (delaying the primary, if necessary, to guarantee the synchronization), assuming we expect such events to be important but relatively rare. Synchronizing on such an event provides a hard guarantee that can prevent the spread of an attack, yet maximizes the amount of time the analysis machine has to catch up with the primary.

8 Related work

Replay facilities in a VMM have been discussed by a number of researchers [4, 9, 35] and used for a variety of purposes. For example, Bressoud and Schneider log non-determinism to support re-execution of a whole machine (OSes and applications) and use this to tolerate fail-stop faults on HP PA-RISC [4]. ReVirt [9] uses VM replay on *x86* systems to enable *ex post facto* analysis for computer forensics. Aftersight uses VM replay for another purpose, which is to enable heavyweight dynamic analysis to be used on realistic workloads without perturbing them. Aftersight is also more flexible than these past VM replay systems because it allows analyses to run in a different environment from the primary, such as a simulator. Aftersight also leverages the fact that replicas can run faster than the primary to make online analysis more practical.

Researchers have suggested using replay implemented at the virtual-machine level or in hardware to conduct various types of offline analyses, such as computer forensics [9, 14], debugging [15, 16, 34], and architectural simulation [35]. Aftersight makes more types of analysis practical by allowing the analysis to run in a simulator, which reduces the cost of context switching between the replaying virtual machine and the analysis code. Analyses like taint analysis, which require frequent switches, are impractical without this capability. Aftersight also extends the use of VM replay to both online and offline

analysis.

Other researchers have sought to run analyses online and in parallel with the original program via software or hardware support. Patil and Fischer proposed running an instrumented “shadow process” in parallel with the original program [21] and used this approach to implement memory safety checks. Speck [19] and SuperPin [31] fork multiple analysis processes from an uninstrumented process, using record/replay to synchronize the analysis processes. Whereas these past approaches can only analyze an application process, Aftersight expands the scope of decoupled analysis to include the operating system and all applications running on a machine. Aftersight also uses a more complete replay system that handles asynchronous interrupts.

Oplinger and Lam leverage proposed hardware support for thread-level speculation (TLS) to enable the original program to run in parallel with monitoring code [20]. They depend on proposed hardware support for TLS to detect data dependencies and rollback the original program if it causes a conflict. Similarly, Zhou, et al. use proposed hardware support for TLS to run memory-monitoring functions in parallel with the original program [36]. In contrast to this prior work, Aftersight requires no hardware support and works on today’s commodity processors. Without support for TLS, current processors cannot quickly fork a new thread. Instead, Aftersight uses virtual-machine replay to continuously mirror the dynamic state of the original program on the analysis machines, thereby making it possible for them to analyze this state on spare processors or cores. Aftersight also includes new optimizations to accelerate the analysis machines by leveraging information generated by the original program.

A recent workshop paper briefly describes a similar approach to executing analyses in parallel with the original program [6]. As with TLS, their system requires hardware support to mirror the dynamic state of the original program onto spare processors by logging detailed data from each instruction, including the instruction counter, type, and input/output identifiers. The large volume of log data slows performance down by 4-10x. In contrast, Aftersight requires no hardware support, and runs with low overhead.

Other research has looked at combining simulators and hypervisors. For example, Ho, et al. [12] allowed switching back and forth between QEMU and Xen, and SimOS allows users to switch between direct execution and detailed simulation [22]. Aftersight differs from these systems by decoupling the analyses from the main workload and allowing both modes to run at the same time. This takes advantage of parallelism in processor cores and eliminates the user-visible overheads of analysis.

Besides decoupled analysis, there are a variety of other ways to reduce the overhead of heavyweight analysis. For example, sampling reduces analysis overhead [7] but can miss relevant events and only works for specific analyses. Hardware solutions [29] can also reduce the perturbation to the main workload caused by analysis, but requiring custom hardware makes this approach less attractive.

9 Conclusions

Dynamic program analysis has a wide range of compelling uses. Unfortunately, powerful analyses typically add substantial overhead which perturbs the workload, so the vast majority of program execution takes place with very little checking. This means that many critical software flaws remain overlooked, when they could be detected during testing, quality assurance, and deployment. Similarly, in operational settings, the high overhead of analysis deters the use of many potentially promising techniques for intrusion detection and prevention.

We have presented *Aftersight*, a system that helps overcome these limitations by decoupling dynamic program analysis from execution through virtual machine replay. This allows analysis to be carried out on the replayed execution, independent of the main workload. This mechanism allows several choices along the safety/performance spectrum, such as synchronous safety, best-effort safety, and offline analysis. Synchronous safety achieves performance comparable to inline analysis, while best-effort safety and offline analysis make it possible to apply slow, expensive analysis techniques on realistic production workloads without perturbing their performance.

We discussed how Aftersight supports the use of different record and replay platforms and the benefits of allowing each to be independently optimized based on their need for performance or extensibility. We presented our prototype of Aftersight, and evaluated it with several online and offline analyses.

Dynamic program analysis is a promising technique for solving many problems. However, without a means of overcoming its performance costs, it will continue to see limited use. In light of the ubiquitous adoption of virtualization technology, we believe decoupled analysis, as demonstrated by Aftersight, offers a promising approach to enabling the use of this technique in a much broader set of applications.

References

- [1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS*, pages 2–13, October 2006.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

- [3] F. Bellard. QEMU, A Fast and Portable Dynamic Translator. In *Proc. USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
- [4] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.
- [5] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.
- [6] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-Based Architectures for General-Purpose Monitoring of Deployed Code. *2006 Workshop on Architectural and System Support for Improving Software Dependability*, October 2006.
- [7] T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *ASPLOS*, October 2004.
- [8] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proc. 1998 SIGMETRICS Symposium on Parallel and distributed tools (SPDT)*, August 1998.
- [9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, pages 211–224, New York, NY, USA, 2002. ACM.
- [10] S. L. Graham, P. B. Kessler, and M. E. McKusick. Gprof: A Call Graph Execution Profiler. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.
- [11] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proc. Winter 1992 USENIX Conference*, pages 125–138, 1992.
- [12] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection Using Demand Emulation. In *EuroSys*, pages 29–41, New York, NY, USA, 2006. ACM Press.
- [13] Intel. IA-32 Intel Architecture Software Developer's Manual. Volumes I, II, and III, 2006.
- [14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP*, pages 91–104, October 2005.
- [15] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Travelling Virtual Machines. In *Proc. USENIX Annual Technical Conference*, pages 1–15, 2005.
- [16] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.
- [17] N. Nethercote and J. Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *PLDI*, pages 89–100, New York, NY, USA, 2007. ACM Press.
- [18] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2005.
- [19] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *ASPLOS*, March 2008.
- [20] J. Oplinger and M. S. Lam. Enhancing Software Reliability using Speculative Threads. In *ASPLOS*, pages 184–196, October 2002.
- [21] H. Patil and C. N. Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *Proc. International Workshop on Automated and Algorithmic Debugging (AADEBUD)*, pages 119–132, May 1995.
- [22] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.
- [23] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, May 2005.
- [24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [25] B. A. Schroeder. On-Line Monitoring: a Tutorial. *IEEE Computer*, 28(6):72–78, June 1995.
- [26] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors With Bit-Precision. In *Proc. USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.
- [27] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flash-back: A light-weight rollback and deterministic replay extension for software debugging. In *Proc. USENIX Technical Conference*, June 2004.
- [28] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX Annual Technical Conference*, pages 1–14, 2001.
- [29] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.
- [30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, pages 181–194, December 2002.
- [31] S. Wallace and K. Hazelwood. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *Proc. 2007 International Symposium on Code Generation and Optimization (CGO)*, pages 209–217, March 2007.
- [32] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*, December 2004.
- [33] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *SIGMETRICS Perform. Eval. Rev.*, 24(1):68–79, 1996.
- [34] M. Xu, R. Bodik, and M. D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *ISCA*, pages 122–135, New York, NY, USA, 2003. ACM Press.
- [35] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proc. 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2007.
- [36] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *ISCA*, June 2004.

Protection Strategies for Direct Access to Virtualized I/O Devices

Paul Willmann, Scott Rixner, and Alan L. Cox

Rice University

{willmann, rixner, alc}@rice.edu

Abstract

Commodity virtual machine monitors forbid direct access to I/O devices by untrusted guest operating systems in order to provide protection and sharing. However, both I/O memory management units (IOMMUs) and recently proposed software-based methods can be used to reduce the overhead of I/O virtualization by providing untrusted guest operating systems with safe, direct access to I/O devices. This paper explores the performance and safety tradeoffs of strategies for using these mechanisms.

The protection strategies presented in this paper provide equivalent inter-guest protection among operating system instances. However, they provide varying levels of intra-guest protection from driver software and incur varying levels of overhead. A simple direct-map strategy incurs the least overhead, providing native-level performance but offering no enhanced protection from misbehaving device drivers within the guest operating system. Additional protection against guest drivers can be achieved by limiting IOMMU page-table mappings to memory buffers that are actually used in I/O transfers. Furthermore, the cost incurred by this limitation can be minimized by aggressively reusing these mappings. Surprisingly, a software-only strategy that does not use an IOMMU at all performs competitively, and sometimes better than, hardware-based strategies while maintaining strict inter-guest isolation.

1 Introduction

In many organizations, the economics of supporting a growing number of Internet-based application services has created a demand for server consolidation. Consequently, there has been a resurgence of interest in ma-

chine virtualization [1, 2, 5, 8, 9, 10, 14, 19, 22]. However, virtualization can impose performance penalties up to a factor of 5 on I/O-intensive workloads [16, 19]. These penalties stem from the overhead of providing shared and protected access to I/O devices by untrusted guest operating systems. Commodity virtualization architectures provide protection in software by forbidding direct access to I/O hardware by untrusted guests—instead, all I/O accesses, both commands and data, are routed through a single software entity that provides both protection and sharing.

Preferably, guest operating systems would be able to directly access I/O devices without the need for the data to traverse an intermediate software layer within the virtual machine monitor [17, 23]. However, if a guest can directly access an I/O device, then it can potentially direct the device to access memory that it is not entitled to via direct memory access (DMA). Therefore, the virtual machine monitor must be able to ensure that guest operating systems do not access each other's memory indirectly through the shared I/O devices in the system. Both I/O memory management units (IOMMUs) [7] and recently proposed software-based methods [23] can provide DMA memory protection for the virtual machine monitor. They do so by preventing guest operating systems from directing I/O devices to access memory that it is not entitled to access, while still allowing the guest to directly access the device.

These DMA protection mechanisms can also be used by a guest operating system to enhance safety and isolation among its own drivers and processes. The state-of-the-art single-use IOMMU-based protection strategy employed by many existing non-virtualized operating systems provides just such a level of enhanced safety. This strategy creates a mapping for each I/O transaction and then destroys that mapping as soon as the transaction completes. In conjunction with IOMMU hardware, the operating system's protection strategy can exert fine-grained control over what portions of memory may be

This work was supported in part by the National Science Foundation under Grant Nos. CCF-0546140 and CNS-0720878 and by gifts from Advanced Micro Devices and Hewlett-Packard. Paul Willmann was supported in part by SFE Technology, Inc.

used in an I/O transaction at any given time.

This paper explores and experimentally compares five different strategies for ensuring memory isolation of untrusted, virtualized operating systems that each have direct access to I/O hardware. These strategies all ensure isolation among OS instances and the virtual machine monitor, but they vary in the level of protection within a particular guest they can support and the overheads they incur. Though upcoming commodity platforms will feature IOMMUs for efficient I/O virtualization, there exists no comprehensive study about how to best leverage IOMMUs, what the tradeoffs are among efficiency and protection for different possible strategies, and what the comparative costs associated with the various protection strategies are.

The first IOMMU-based strategy is based on state-of-the-art strategies and uses single-use I/O memory mappings that are created before each I/O operation and immediately destroyed after each I/O operation. The second IOMMU-based strategy is introduced in this paper and uses shared I/O memory mappings that can be reused by multiple, concurrent I/O operations. The third IOMMU-based strategy is also introduced in this paper and uses persistent I/O memory mappings that can be reused. The fourth IOMMU-based strategy uses a static direct map of the guest's physical memory to isolate that guest's I/O transactions. Finally, the software-based strategy is based on previous work [23] and uses VMM-managed DMA descriptors that can only be used for one I/O operation.

The comparison of these five strategies yields several insights. First, all five strategies provide equivalent inter-guest protection among OS instances. However, the strategies support differing levels of protection within a particular guest (*intra-guest* protection). For example, the direct-map strategy incurs almost no performance overhead but supports no intra-guest protection. Conversely, the single-use strategy provides the maximum possible intra-guest protection, but it imposes the largest performance penalty. Second, there is significant opportunity to reuse IOMMU mappings, which can reduce protection overheads. Multiple concurrent I/O operations are able to share the same mappings often enough that there is a noticeable decrease in the overhead of providing protection. Sharing mappings only among concurrent I/O operations provides the same level of intra-guest protection as the single-use strategy but with less overhead. Relaxing this intra-guest protection guarantee by allowing mappings to persist so that they can be reused in future I/O operations can significantly decrease this overhead, allowing the guest to achieve performance levels very close to that of the direct-map strategy while still maintaining some amount of intra-guest protection. Finally, the software-based protec-

tion strategy performs competitively with several of the better-performing IOMMU-based strategies while maintaining strong inter-guest protection guarantees and enabling intra-guest protection capabilities.

The next section provides background on how I/O devices access main memory and the possible memory protection violations that can occur when doing so. Sections 3 and 4 discuss the four IOMMU-based protection strategies and the one software-based protection strategy. Section 5 then describes the protection properties afforded by the five strategies. Section 6 discusses IOMMU hardware architectures. Section 7 describes the experimental methodology and Section 8 evaluates the protection strategies. Section 9 then describes related work and Section 10 concludes the paper.

2 Background

Modern server I/O devices, including disk and network controllers, utilize direct memory access (DMA) to move data between the host's main memory and the device's on-board buffers. The device uses DMA to access memory independently of the host CPU, so such accesses must be controlled and protected. To initiate a DMA operation, the device driver within the operating system creates *DMA descriptors* that refer to regions of memory. Each DMA descriptor typically includes an address, a length, and a few device-specific flags. In commodity x86 systems, devices lack support for virtual-to-physical address translation, so DMA descriptors always contain physical addresses for main memory. Once created, the device driver passes the descriptors to the device, which will later use the descriptors to transfer data to or from the indicated memory regions autonomously. When the requested I/O operations have been completed, the device raises an interrupt to notify the device driver.

For example, to transmit a network packet, the network interface's device driver might create two DMA descriptors. The first descriptor might point to the packet headers and the second descriptor might point to the packet payload. Once created, the device driver would then notify the network interface that there are new DMA descriptors available. The precise mechanism of that notification depends on the particular network interface, but typically involves a programmed I/O operation to the device telling it the location of the new descriptors. The network interface would then retrieve the descriptors from main memory using DMA—if they were not written to the device directly by programmed I/O. The network interface would then retrieve the two memory regions that compose the network packet and transmit them over the network. Finally, the network interface would interrupt the host to indicate that the packet has been transmitted. In practice, notifications from the device

driver and interrupts from the network interface would likely be aggregated to cover multiple packets for efficiency.

Three potential memory access violations can occur on every I/O transfer initiated using this DMA architecture:

1. The device driver could create a DMA descriptor with an incorrect address (a “bad-address” fault).
2. The operating system could repurpose the memory referenced by a DMA descriptor, or the device driver could later reuse a valid DMA descriptor without permission (an “invalid-use” fault).
3. The device itself could initiate a DMA transfer to a memory address not referenced by the DMA descriptor (a “bad-device” fault).

These violations could occur either because of failures or because of malicious intent. However, as devices are typically not user-programmable, the last type of violation is only likely to occur as a result of a device failure.

In a non-virtualized environment running on commodity x86 hardware, the operating system is solely responsible for preventing “bad-address” and “invalid-use” violations. This requires the operating system to trust the device driver to create correct DMA descriptors using only physical memory addresses of buffers that have been pinned by the OS. In practical terms, however, trusting the driver can be disastrous in terms of system stability. For example, nearly 85% of all system crashes of the Windows XP operating system are caused by drivers [20]. As will be discussed, operating systems running on platforms that feature IOMMUs can leverage those hardware capabilities to isolate device drivers and explicitly authorize the I/O transactions requested by the driver, thus reducing the trust requirement for the driver. Regardless, a failure of the operating system to prevent these memory access violations could potentially result in system failure.

In a virtualized environment, however, the virtual machine monitor cannot trust the guest operating systems to prevent these memory access violations, as a memory access violation incurred by one guest operating system can potentially harm other guest operating systems or even bring down the whole system. Therefore, a virtual machine monitor requires mechanisms to prevent one guest operating system from intentionally or accidentally directing an I/O device to access the memory of another guest operating system. The only way that would be possible is via either a “bad-address” or “invalid-use” violation. Depending on the reliability of the I/O devices, it may also be desirable to try to prevent “bad-device” violations as well (although it is frequently not possible

to protect against a misbehaving device, as will be discussed in Section 5). The following sections describe mechanisms and strategies for preventing these memory access violations.

3 IOMMU-based Protection

A VMM can utilize an IOMMU to help provide DMA memory protection when allowing direct access to I/O devices. Whereas a virtual memory management unit enforces access control and provides address translation services for a processor as it accesses memory, an IOMMU enforces access control and provides address translation services for an I/O device as it accesses memory.

In general, the structures defined by IOMMUs for expressing access control and address translation are fairly similar to those defined by virtual memory management units. Typically, the VMM or OS maintains one or more page table structures for use by the IOMMU. There are, however, some differences. The VMM or OS must also maintain another structure for the IOMMU that maps each I/O device to one of the page tables. Although every device must have an entry in this mapping, not every device must have its own page table. If two or more devices are entitled to access the same memory, then they can share a single page table. On every memory access by an I/O device the IOMMU consults the I/O device’s designated page table to determine if the access should be allowed or blocked.

Regardless of the page-table organization, all IOMMU-based systems require that a valid IOMMU mapping exists for each host memory buffer to be used in an upcoming DMA descriptor. Otherwise, the DMA descriptor will refer to a region unmapped by the IOMMU, and the I/O transaction will fail. The following subsections present four strategies for using an IOMMU to provide DMA memory protection in a VMM. The strategies primarily differ in the extent to which IOMMU mappings are allowed to be reused. The underlying IOMMU hardware architectures that may be used to implement these strategies are discussed in more detail in Section 6

3.1 Single-use Mappings

A common strategy for managing an IOMMU is to create a single-use mapping for each I/O transaction. The Linux DMA-Mapping interface, for example, implements a single-use mapping strategy. Ben-Yehuda, *et al.* also explored a single-use mapping strategy in the context of virtual machine monitors [7]. In such a single-use strategy, the driver must ensure that a new IOMMU mapping

is created for each DMA descriptor. Even, when separate DMA descriptors refer to the same physical page, the single-use strategy always creates distinct IOMMU mappings for each descriptor. Each IOMMU mapping is destroyed once the corresponding I/O transaction has completed. In a virtualized system, the trusted virtual machine monitor is responsible for creating and destroying IOMMU mappings at the driver's request. If the VMM does not create the mapping, either because the driver did not request it or because the request referred to memory not owned by the guest, then the device will be unable to perform the corresponding DMA operation.

To carry out an I/O transaction using a single-use mapping strategy, the virtual machine monitor (VMM), untrusted guest operating system (GOS), and the device (DEV) carry out the following steps:

1. GOS: The guest OS requests an IOMMU mapping for the memory buffer involved in the I/O transaction.
2. VMM: The VMM validates that the requesting guest OS has appropriate read or write permission for each memory page in the buffer to be mapped.
3. VMM: The VMM marks the memory buffer as "in I/O use", which prevents the buffer from being reallocated to another guest OS during an I/O transaction.
4. VMM: The VMM creates one or more IOMMU mappings for the buffer. As with virtual memory management units, one mapping is usually required for each memory page in the buffer.
5. GOS: The guest OS creates a DMA descriptor with the IOMMU-mapped address that was returned by the VMM.
6. DEV: The device carries out its I/O transaction as directed by the DMA descriptor and it notifies the driver upon completion.
7. GOS: The driver requests destruction of the corresponding IOMMU mapping(s).
8. VMM: The VMM validates that the mappings belong to the guest OS making the request.
9. VMM: The VMM destroys the IOMMU mappings.
10. VMM: The VMM clears the "in I/O use" marker associated with each memory page referred to by the recently-destroyed mapping(s).

3.2 Shared Mappings

Rather than creating a new IOMMU mapping for each new DMA descriptor, it is possible to share a mapping among DMA descriptors so long as the mapping points to the same underlying memory page and remains valid. Unlike the single-use strategy, the shared-mapping strategy detects when a valid IOMMU mapping to a memory page already exists and reuses that mapping rather than generating a new one. Sharing IOMMU mappings is advantageous because it avoids the overhead of creating and destroying a new mapping for each I/O request. In practical terms, this sharing can happen when an application repeats the same I/O message or when an application sends or receives small I/O messages that reside in the same memory page.

To implement sharing, the guest operating system must keep track of which IOMMU mappings are currently valid, and it must keep track of how many pending I/O requests are currently using the mapping. To protect a guest's memory from errant device accesses, an IOMMU mapping should be destroyed once all outstanding I/O requests that use the mapping have been completed. Though the untrusted guest operating system has responsibilities for carrying out a shared-mapping strategy, it need not function correctly to ensure isolation among operating systems, as is discussed further in Section 5.

To carry out a shared-mapping strategy, the guest OS and the VMM perform many of the same steps that are required by the single-use strategy. The shared-mapping strategy differs at the initiation and termination of an I/O transaction. Before step 1 would occur in a single-use strategy, the guest operating system first queries a table of known, valid IOMMU mappings to see if a mapping for the I/O memory buffer already exists. If so, the driver uses the previously established IOMMU-mapped address for a DMA descriptor, and then passes the descriptor to the device, in effect skipping steps 1–4. If not, the guest and VMM follow steps 1–4 to create a new mapping. Whether a new mapping is created or not, before step 5, the guest operating system increments its own reference count for the mapping. This reference count is separate from the reference count maintained by the VMM.

Steps 5 and 6 then proceed as in the single-use strategy. After these steps have completed, the driver calls the guest operating system to decrement its reference count. If the reference count is zero, no other I/O transactions are in progress that are using this mapping, so the guest calls the VMM to destroy the mapping, as in steps 7–10 of the single-use strategy. Otherwise, the IOMMU mapping is still being used by another I/O transaction within the guest OS, so steps 7–10 are skipped.

3.3 Persistent Mappings

IOMMU mappings can further be reused by allowing them to persist even after all I/O transactions using the mapping have completed. Compared to a shared mapping strategy, such a persistent mapping strategy attempts to further reduce the overhead associated with creating and destroying IOMMU mappings inside the VMM. Whereas sharing exploits reuse among mappings only when a mapping is being actively used by at least one I/O transaction, persistence exploits temporal reuse across periods of inactivity.

The infrastructure and mechanisms for implementing a persistent mapping strategy are similar to those required by a shared mapping strategy. The primary difference is that the guest operating system does not request that mappings be destroyed after the I/O transactions using them complete. Therefore, in contrast to the shared mapping strategy, when the guest's reference count is decremented after step 6, the I/O transaction is complete and steps 7–10 are always skipped. This should dramatically reduce the number of potentially costly invocations of the VMM.

Eventually, it is possible that all of a guest's memory would become mapped using this strategy. Compared to a shared mapping strategy, this increases the guest's exposure to intra-guest protection violations, which will be discussed in Section 5.2. To limit this exposure, the guest operating system can implement a reclamation policy that eventually removes mappings that are not currently in use by an I/O operation. For example, the simple reclamation policy that is used in the experiments of this paper limits the total number of mappings. Once this total is reached, a mapping that is not currently in use would have to be destroyed before a new mapping can be created.

3.4 Direct Mappings

To allow maximum reuse of IOMMU mappings and to further reduce runtime overhead, it is possible to permanently map the entire physical address space of the guest operating system. Such a strategy is sometimes referred to as a *direct map*, because this arrangement creates a one-to-one mapping between IOMMU entries and physical pages for each physical page owned by the guest operating system.

4 Software-based Protection

IOMMU-based protection strategies enforce safety even when untrusted software provides unverified DMA descriptors directly to hardware, because the DMA operations generated by any device are always subject to later

validation. However, an IOMMU is not necessary to ensure full isolation among untrusted guest operating systems, even when they use DMA-capable hardware that directly reads and writes host memory. Rather than relying on hardware to perform late validation during I/O transactions, a lightweight software-based system performs early validation of DMA descriptors before they are used by hardware. The software-based strategy also must protect validated descriptors from subsequent unauthorized modification by untrusted software, thus ensuring that all I/O transactions operate only on buffers that have been approved by the VMM. This software-based strategy was previously introduced as a means for ensuring DMA memory protection by untrusted guest operating systems that have concurrent direct access to a prototype network interface [23].

The runtime operation of a software-based protection strategy works much like a single-use IOMMU-based strategy, since both validate permissions for each I/O transaction. Whereas the single-use IOMMU-based strategy uses the VMM to create IOMMU mappings for each transaction, software-based I/O protection creates the actual DMA descriptor. The descriptor is valid only for the single I/O transaction. Unlike an IOMMU-based system, an untrusted guest OS's driver must first register itself with the VMM during initialization. At that time, the VMM takes ownership of the driver's DMA descriptor region and the driver's status region, revoking write permissions from the guest. This prevents the guest from independently creating or modifying DMA descriptors, or modifying the status region. Finally, the VMM must prevent the guest from changing the descriptor and status regions. This can be accomplished by only mapping the device's configuration registers into the VMM's address space, and not into the guests' address spaces.

After initialization, the operation of the software-based strategy is similar to the single-use IOMMU-based strategy outlined in Section 3.1. Steps 1–3 of a software-based strategy are nearly identical, with the exception that the Guest OS is requesting a DMA descriptor, not an IOMMU mapping, in Step 1. In step 4, the VMM creates a DMA descriptor in the write-protected DMA descriptor region, obviating the OS's role in step 5. The device carries out the requested operation using the validated descriptor, as in step 6, and because the descriptor is write-protected, the untrusted guest cannot modify the descriptor and thus cannot induce a transaction that has not been explicitly authorized by the VMM. When the device signals completion of the transaction, the VMM inspects the device's state (which is usually written via DMA back to the host) to see which DMA descriptors have been used. The VMM then processes those completed descriptors, as in step 10, permitting the associated guest memory buffers to be reallocated.

	Inter-Guest			Intra-Guest		
	Bad Address	Invalid Use	Bad Device	Bad Address	Invalid Use	Bad Device
Direct-map	x	x	x			
Single-use	x	x	x	x		x
Shared	x	x	x	x		x
Persistent	x	x	x	x		
Software	x	x		x	x	

Table 1: Types of protection supported by the different DMA protection strategies.

In contrast to the IOMMU-based strategies, the software-based strategy requires that the VMM actually insert DMA descriptors to the I/O device. This requires that the VMM know the method of insertion (*i.e.*, programmed I/O or DMA) and the structure of the DMA descriptor for a particular device. Furthermore, the VMM must be able to determine the descriptor state of the device. Descriptor state specifies whether or not the device can accept more descriptors and indicates when previously posted descriptors have been processed. Though these device-specific requirements inherently require some device-specific methods in the VMM, the general method of software protection described here apply to DMA-capable devices in general. As described in greater detail in [23], the implementation for software-based DMA protection described here applies to devices that organize their DMA descriptors in contiguous rings, which includes many high-performance devices.

5 Protection Properties

The protection strategies presented in Sections 3 and 4 can be used to prevent the memory access violations presented in Section 2. Those violations can happen by creating a DMA descriptor using a bad address, by repurposing and reusing a DMA descriptor after its initial use, or by suffering a fault inflicted by a malfunctioning I/O device. These violations can occur both across multiple guests (inter-guest) and within a single guest (intra-guest). A virtual machine monitor must, at minimum, provide inter-guest protection in order to operate reliably. A guest operating system may additionally benefit if the system hardware or the virtual machine monitor can be used to help provide intra-guest protection. This section describes the protection properties of the five previously presented protection strategies. Table 1 summarizes these faults and shows which strategy prevents what faults in both the inter-guest and intra-guest cases.

5.1 Inter-Guest Protection

Perhaps surprisingly, all five strategies provide equivalent inter-guest protection against “bad-address” and

“invalid-use” faults. In all of the IOMMU-based strategies, if the device driver creates a DMA descriptor that refers to memory that is not owned by that guest operating system, the device will be unable to perform that DMA, as no IOMMU mapping will exist. The only requirement to maintain this protection is that the VMM must never create an IOMMU mapping for a guest that does not refer to that guest’s memory. Similarly, only the VMM can repurpose memory to another guest, so as long as it does not do so while there is an existing IOMMU mapping to that memory, inter-guest “invalid-use” faults can never occur. The software-based approach provides exactly the same guarantees by only allowing the VMM to create DMA descriptors. Therefore, these strategies allow the VMM to provide protection.

“Bad-device” faults are more difficult to prevent. If the device is shared among multiple guest operating systems, then no strategy can prevent this type of fault. For example, if a network interface is allowed to receive packets for two guest operating systems, the VMM cannot prevent the device from sending the traffic destined for one guest to the other. This is one simple example of many of the many problems that a shared device can cause.

However, if a device is privately assigned to a single guest operating system, the IOMMU-based strategies can be used to provide protection against faulty device behavior. In this case, the VMM simply has to ensure that there are only IOMMU mappings to the guest that is assigned the device. In this manner, all four IOMMU-based strategies can protect against such a fault. However, the software-based strategy cannot provide this level of protection. Though DMA descriptors are validated by the VMM to ensure that they only point to memory for which the associated guest has appropriate permissions, there is no software-only mechanism capable of stopping the device from simply ignoring the DMA descriptor and accessing any physical memory.

5.2 Intra-Guest Protection

As is shown in Table 1, the five protection strategies discussed in this paper vary significantly according to how they may be used by the guest OS to prevent intra-guest

faults of the types listed in Section 2. In order to prevent a driver from creating a DMA descriptor with the wrong address or using memory that has been repurposed by the guest OS (i.e., errors of the first two types), the OS must implement its own isolation strategy for inspecting and verifying each I/O transaction. Typically, operating systems designed for commodity platforms simply trust that the driver will request a valid, current address for an I/O transaction.

To enable the OS to protect itself from drivers that may construct DMA descriptors using bad intra-guest addresses, the OS must be able to act as the gate-keeper to I/O translations and thus DMA addresses. Unlike the other four strategies, the direct-map strategy ensures that all of a guest's memory is mapped in the IOMMU at any given time, and thus the driver does not have to request specific permission for I/O memory from the OS. In this direct-map case, it is possible for the driver to create a valid DMA descriptor to an arbitrary region of the guest's memory. This I/O transaction would succeed with the blessing of the IOMMU so long as the memory location is somewhere within the guest's memory, even though the OS may have never approved use of this location for I/O. In all the other strategies, at least one specific request for memory by the driver is required before the OS will approve construction of the necessary IOMMU mapping, and hence, all the other strategies can protect against intra-guest "bad-address" faults.

Once the first request to create the IOMMU mapping has happened, however, none of the IOMMU-based strategies can prevent a driver from invalidly reusing that same mapping for a subsequent I/O transaction. In these strategies, the driver is responsible for informing the OS when it is done with an IOMMU mapping. Even if the OS was modified to automatically revoke an IOMMU mapping when it detected the completion of a corresponding I/O event (as in the completion of a `sendfile()` operation or the `free()` of an `skbuff`), the driver could still invalidly reuse a mapping after the original I/O event finished, but before the OS could intervene to terminate the IOMMU mapping. In the software strategy, however, the VMM automatically detects when the device has completed a specific I/O transaction and ensures that individual DMA descriptors can never be reused. Thus, the software-based DMA protection strategy can be used by an operating system to detect and then prevent "invalid-use" faults.

Preventing "bad-device" faults from corrupting an individual guest's memory requires that the device can only access that guest's memory while a valid I/O transaction is in-flight and has been authorized by the OS. The direct-map strategy is incapable of preventing these faults because it permanently maps all of a guest's memory for I/O, and thus it may be used for I/O at any given

time. The persistent strategy allows IOMMU mappings to exist in a valid state even after I/O transactions have completed, and thus in this time period, it is possible for a device fault to access one of those mapped locations and corrupt memory. As in the inter-guest case, the software-based mechanism has no hardware enforcement mechanism that could prevent the device from initiating an invalid transfer. Conversely, both the single-use and shared-mapping strategies are specifically designed to only permit valid IOMMU mappings to exist during active I/O transactions, and hence they both guard against device faults.

6 IOMMU Architectures

As discussed in Section 3, an IOMMU performs memory translation and protection for I/O devices. For each direct memory access (DMA) performed by an I/O device, the IOMMU validates that the device is allowed to access that memory and translates the memory address appropriately. If the device is not allowed to access that memory or there is no valid translation, then the IOMMU terminates the DMA transaction.

A graphics address relocation table (GART) provides similar memory translation functionality for I/O devices. A GART translates memory addresses within a contiguous range in the physical address space, called the GART *aperture*. Unlike an IOMMU, only addresses that fall within this aperture are translated by the GART. This translation functionality is typically used by graphics software libraries to simplify memory accesses performed by the graphics card. Operating systems also use GART hardware to provide translation of addresses for devices that only support 32-bit addressing on 64-bit platforms. Though the GART translates memory addresses in a similar manner to an IOMMU, the GART does not protect memory from the device. Devices can still access all physical memory directly using addresses outside of the GART aperture.

Table 2 shows the performance differences between the AMD Opteron with an integrated GART and two modern IOMMU platforms, the IBM Calgary platform and the Intel VT-d architecture. The table shows the average cost, in processor cycles, to update an I/O page table (PT) entry, to flush the platform's I/O translation lookaside buffer (IOTLB) which caches translations, and to both update an I/O page table entry and then immediately flush the IOTLB. The Calgary, VT-d and GART platforms feature processors operating at 2.5, 2.66, and 2.4 GHz, respectively. The performance differences among the platforms arise directly from the differing page table organizations and IOTLB overheads.

As the table shows, the Calgary platform has the highest overhead to install or modify a translation for a single

Platform	I/O PT Update	IOTLB Flush	Update & Flush
IBM Calgary IOMMU	673	10207	10887
Intel VT-d IOMMU	991	1217	2213
AMD GART	27	486	579

Table 2: Microbenchmarks examining costs associated with modern translation hardware, in processor cycles.

memory page. This is primarily caused by its unusually high cost for flushing the system's IOTLB, which is discussed at length by Ben-Yehuda *et al.* [6]. The Intel VT-d architecture is more efficient, but its multilevel page table and its IOTLB are still more expensive to update than the flat page table organization of the GART and its simpler IOTLB interface.

The Opteron's GART was used to evaluate the protection strategies presented in this paper for two reasons. First, at the time this research began, there were no IOMMUs available for x86-based systems—the AMD Opteron with a GART was the only suitable device. Second, as Table 2 shows, the currently available IOMMUs for x86-based systems, Calgary and VT-d, have higher overheads than the GART. The only implication of this choice is that the strategies that perform more frequent mappings, single-use and shared, perform better than they would with a higher-overhead IOMMU. In fact, Ben-Yehuda, *et al.* found that the single-use strategy had higher overhead than found in this paper [7].

7 Experimental Setup

The protection strategies described in Sections 3 and 4 were evaluated on a system with an AMD Opteron 250 processor. The Opteron's GART is used to model the behavior and functionality of IOMMU hardware, as described in the previous section. The IOMMU- and software-based protection strategies are implemented in the open source Xen 3 virtual machine monitor [5]. Xen differs from many virtualization systems in that it exposes host physical addresses to the guest OS. In particular, the guest OS, and not the VMM, is responsible for translating between pseudo-physical addresses that are used at most levels of the guest OS and host physical addresses that are used at the device level. This does not, however, fundamentally change the implementation of the various protection strategies.

We evaluate these strategies on a variety of network-intensive workloads, including a TCP stream microbenchmark, a voice-over-IP (VoIP) server benchmark, and a static-content web server benchmark. The stream microbenchmark either transmits or receives bulk data over a TCP connection to a remote host. The VoIP benchmark uses the OpenSER server. In this benchmark,

OpenSER acts as a SIP proxy and 50 clients simultaneously initiate calls as quickly as possible. The web server benchmark uses the lighttpd web server to host static HTTP content. In this benchmark, 32 clients simultaneously replay requests from various web traces as quickly as possible. Three web traces are used in this study: "CS", "IBM", and "WC". The CS trace is from the Rice University computer science departmental web server and has a working set of 1.2 GB of data. The IBM trace is from an IBM web server and has a working set of 1.1 GB of data. The WC trace is from the 1998 World Cup soccer web server and has a working set of 100 MB of data. For all benchmarks, the client machine is directly connected to the server without the use of a switch, and the client is monitored to ensure that it is never saturated. Hence, it is assured that the server machine is always the bottleneck. Each benchmark (TCP stream, VoIP, and each web benchmark) is tested using the specified strategy a minimum of 5 times each. The performance reported is average performance, because there is no significant variance across runs.

The server under test has two Gigabit Ethernet network interface cards and features DDR 400 DRAM. The network interfaces are publicly available prototypes that are user-programmable, support shared direct access by virtual machines, and support line-rate Gigabit Ethernet speeds [18]. For each configuration except the direct-map strategy, a single unprivileged guest operating system has 1.4 GB of memory, and the IOMMU-based strategies employ 512 MB of physical GART address space for remapping (which corresponds to 131,072 unique mappings). For the Direct-map strategy, we use a guest operating system with 512 MB of memory. We simplify the implementation of the Direct-map strategy to its minimum possible overhead by pre-mapping the entire guest's physical memory space permanently at boot-time. Hence, this model represents the minimum possible I/O overhead on this platform, since no mappings are created, destroyed, or modified during the experiments. Its limited memory footprint, however, prevents a fair evaluation of web-based workloads that have a working set larger than 512 MB, including the IBM and CS traces, and hence those benchmarks are not evaluated using the Direct-map strategy.

In each benchmark, direct access for the guest to the hardware is granted only for the network interface cards. Because the guest's memory allocation is large enough to hold each benchmark and its corresponding data set, other I/O is insignificant. For the web-based workloads, the guest's buffer cache is warmed prior to the tests.

Protection Strategy	CPU %		Reuse (%)		HC/ DMA
	Total	Prot.	TX	RX	
Stream Transmit					
None	41	0	N/A	N/A	0
Direct-map	41	0	N/A	N/A	0
Single-use	64	23	N/A	N/A	.88
Shared	58	17	39	0	.55
Persistent	43	2	100	100	0
Software	56	15	N/A	N/A	.90
Stream Receive					
None	53	0	N/A	N/A	0
Direct-map	54	0	N/A	N/A	0
Single-use	79	26	N/A	N/A	.37
Shared	73	20	39	0	.10
Persistent	59	5	100	100	0
Software	64	11	N/A	N/A	.39

Table 3: TCP Stream Profile.

8 Evaluation

Network server applications can stress network I/O in different ways, depending on the characteristics of the application and its workload. Applications may generate large or small network packets, and may or may not utilize zero-copy I/O. For an application running on a virtualized guest operating system, these network characteristics interact with the I/O protection strategy implemented by the VMM. Consequently, the efficiency of the I/O protection strategy can affect application performance in different ways. Furthermore, the application’s behavior for a given workload can directly affect the amount of mapping reuse that is exploitable by a given strategy. We first provide an overview of performance and efficiency under several different network workloads, and then we discuss the sources of mapping reuse for the different workloads.

For all applications, we evaluate the five protection strategies presented earlier, and we compare each to the performance of a system lacking any I/O protection at all (“None”). “Single-use”, “Shared”, “Persistent”, and “Direct-map” all use an IOMMU to enforce protection, using either single-use, shared-mapping, persistent-mapping, or direct-mapping strategies, respectively, as described in Section 3. “Software” uses software-based I/O protection, as described in Section 4.

8.1 TCP Stream

A TCP stream microbenchmark either transmits or receives bulk TCP data and thus isolates network I/O performance. This benchmark does not use zero-copy I/O. Table 3 shows the CPU efficiency and overhead associated with each protection mechanism when streaming data over two network interfaces. The table shows the

total percentage of CPU consumed while executing the benchmark and the percentage of CPU spent implementing the given protection strategy. The table also shows the percentage of times a buffer to be used in an I/O transaction (either transmit or receive) already has a valid IOMMU mapping that can be reused. Finally, the table shows the number of VMM invocations, or hypercalls (HC), required per DMA descriptor used by the network interface driver.

When either transmitting or receiving, all of the strategies achieve the same TCP throughput (1865 Mb/s transmitting, 1850 Mb/s receiving), but they differ according to how costly they are in terms of CPU consumption. The single-use protection strategy is the most costly, with its repeated construction and destruction of IOMMU mappings consuming 23% of total CPU resources for transmit and 26% for receive. The shared strategy reclaims some of this overhead through its sharing of in-use mappings, though this reuse only exists for transmitted packets (data in the transmit-stream case, TCP ACK packets in the receive case). The lack of reuse for received packets is caused by the paravirtualized (PV) Linux buffer allocator, which dedicates an entire 4 KB page for each receive buffer, regardless of the buffer’s actual size. This over-allocation is an artifact of the PV-Linux I/O architecture, which was designed to remap received packets to transfer them between guest operating systems.

Regardless, the persistent strategy achieves 100% reuse of mappings, as the small number of persistent mappings that cover network buffers essentially become permanent. This further reduces overhead relative to single-use and shared. Notably, the number of hypercalls per DMA operation rounds to zero. However, detailed L2 cache profile statistics (not shown in the table) reveals that management of the persistent mappings—mapping lookup and reclamation, as described in Section 3.3—incurs additional overhead in the processor’s memory system. This consumes 2% of the processor resources for the transmit-based workload and 5% for the receive-based workload. The direct-map strategy does not require any protection management at runtime and has the same measured CPU utilization as the “None” case for the transmit case. However, the direct-map strategy incurs a small overhead in the receive case. This represents the measured overhead on the system of simply using the GART for I/O transactions rather than using non-translated addresses. However, through extensive reuse of existing IOMMU mappings, the persistent-mapping strategy achieves nearly the same efficiency as the direct-map case.

Surprisingly, the overhead incurred by the software-based technique is noticeably less than the IOMMU-based shared-mapping and single-use strategies. The software-based technique certainly requires far more hy-

Protection Strategy	Calls/ Sec.	CPU % Prot.	Reuse (%)		HC/ DMA
			TX	RX	
None	3005	0	N/A	N/A	0
Direct-map	2997	0	N/A	N/A	0
Single-use	2790	6.1	N/A	N/A	.68
Shared	2835	6.0	4	0	.65
Persistent	2997	0.1	100	100	0
Software	2895	3.5	N/A	N/A	.67

Table 4: OpenSER Profile.

percalls per DMA than the IOMMU-based strategies. The cost of those VMM invocations and the associated page-verification operations is similar to the cost of inspecting mapping requests for shared- and single-use strategies. However, the software strategy does not incur the additional overhead of flushing the IOMMU's IOTLB via a programmed-I/O write, as is required with the other strategies whenever a group of changes to an I/O page table must be committed.

8.2 VoIP Server

Table 4 shows the performance and overhead profile for the OpenSER VoIP application benchmark for the various protection strategies. The OpenSER benchmark is largely CPU-intensive and therefore only uses one of the two network interface cards. Though the strategies rank similarly in efficiency for the OpenSER benchmark as in the TCP Stream benchmark, Table 4 shows one significant difference with respect to reuse of IOMMU mappings. Whereas the shared strategy was able to reuse mappings 39% of the time for transmit packets under the TCP Stream benchmark, OpenSER sees only 4% reuse. Unlike typical high-bandwidth streaming applications, OpenSER only sends and receives very small TCP messages in order to initiate and terminate VoIP phone calls. Consequently, the shared strategy provides only a minimal efficiency and performance improvement over the high-overhead single-use strategy for the OpenSER benchmark, indicating that sharing alone does not provide an efficiency gain for applications that are heavily reliant on small messages.

8.3 Web Server

Table 5 shows the performance, overhead, and sharing profiles of the various protection strategies when running a webserver under each of three different trace workloads, "CS", "IBM", and "WC". As in the TCP Stream and OpenSER benchmarks, the different strategies rank identically among each other in terms of performance and overhead. Note that the direct-map strategy is evaluated only for the "WC" trace, since it is

Protection Strategy	HTTP Mbps	CPU % Prot.	Reuse (%)		HC/ DMA
			TX	RX	
CS Trace					
None	1336	0	N/A	N/A	0
Single-use	1142	18.2	N/A	N/A	.66
Shared	1162	16.3	40	0	.42
Persistent	1292	3.3	100	100	0
Software	1212	9.1	N/A	N/A	.67
IBM Trace					
None	359	0	N/A	N/A	0
Single-use	322	8.5	N/A	N/A	.70
Shared	322	8.3	22	0	.58
Persistent	350	1.3	100	100	0
Software	326	4.5	N/A	N/A	.71
WC Trace					
None	714	0	N/A	N/A	0
Direct-map	697	0	N/A	N/A	0
Single-use	617	11.8	N/A	N/A	.68
Shared	619	11.1	30	0	.50
Persistent	681	1.8	100	100	0
Software	632	5.9	N/A	N/A	.69

Table 5: Web Server Profile Using write().

the only web trace whose workload will fit entirely within the direct-map configuration's smaller guest operating system memory allocation. Each of the different traces generates messages of different sizes and requires different amounts of web-server compute overhead. For the write()-based implementation of the web server, however, the server is always completely saturated for each workload shown. "CS" is primarily network-limited, generating relatively large response messages with an average HTTP message size of 34 KB. "IBM" is largely compute-limited, generating relatively small HTTP responses with an average size of 2.8 KB. "WC" lies in between, with an average response size of 6.7 KB. As the table shows, the amount of reuse exploited by the shared strategy is dependent on the average HTTP response being generated. Larger average messages lead to larger amounts of reuse for transmitted buffers under the shared strategy. Though larger amounts of reuse slightly reduce the CPU overhead for the shared strategy relative to the single-use strategy, the reuse is not significant enough under these workloads to yield significant performance benefits.

As in the other benchmarks, receive buffers are not subject to reuse with the shared-mapping strategy. Regardless of the workload, the persistent strategy is 100% effective at reusing existing mappings as the mappings again become effectively permanent. As in the other benchmarks, the software-based strategy achieves application performance consistently between the shared and persistent IOMMU-based strategies.

Protection Strategy	HTTP Mbps	CPU %		Reuse (%)			HC/DMA
		Idle	Prot.	TX Hdr.	TX File	RX	
CS Trace							
None	1378	35.0	0	N/A	N/A	N/A	0
Single-use	1291	7.0	27.6	N/A	N/A	N/A	.37
Shared	1330	17.0	17.7	82	72	0	.17
Persistent	1363	28.0	6.7	100	96	100	.02
Software	1351	21.0	13.7	N/A	N/A	N/A	.37
IBM Trace							
None	475	0	0	N/A	N/A	N/A	0
Single-use	403	0	14.0	N/A	N/A	N/A	.43
Shared	413	0	12.3	34	50	0	.35
Persistent	455	0	2.4	100	99	100	0
Software	422	0	6.2	N/A	N/A	N/A	.43
WC Trace							
None	961	0	0	N/A	N/A	N/A	0
Direct-map	953	0	0	N/A	N/A	N/A	0
Single-use	760	0	19.9	N/A	N/A	N/A	.39
Shared	796	0	16.0	53	62	0	.27
Persistent	914	0	2.7	100	100	100	0
Software	833	0	8.7	N/A	N/A	N/A	.40

Table 6: Web Server Profile Using Zero-Copy `sendfile()`.

For all of the previous workloads, the network application utilized the `write()` system call to send any data. Consequently, all buffers that are transmitted to the network interface have been allocated by the guest operating system's network-buffer allocator. Using the zero-copy `sendfile()` interface, however, the guest OS generates network buffers for the packet headers, but then appends the application's file buffers rather than copying the payload. This interface has the potential to change the amount of reuse exploitable by a protection strategy, because data reused by the application can translate to reused IOMMU mappings. Using `sendfile()`, the packet-payload footprint for IOMMU mappings is no longer limited to the number of internal network buffers allocated by the OS, but instead is limited only by the size of physical memory allocated to the guest.

Table 6 shows the performance, efficiency, and sharing profiles for the different protection strategies for web-based workloads when the server uses `sendfile()` to transmit HTTP responses. Note that for the "CS" trace, the host CPU is not completely saturated, and so the CPU's idle time percentage is nonzero. This idle time is useful as a means to compare efficiency. For the other traces, the CPU is completely saturated. The table separates reuse statistics for transmitted buffers according to whether or not the buffer was a packet header or packet payload. As compared to Table 5, Table 6 shows that the shared strategy is more effective overall at exploiting reuse using `sendfile()` than with `write()`. Consequently, the shared strategy gives a larger performance

and efficiency benefit relative to the single-use strategy when using `sendfile()`. Table 6 also shows that the persistent strategy is highly effective at capturing file reuse, even though the total working-set size of the "CS" and "IBM" traces are each more than twice as large as the 512 MB mapping space afforded by the GART. As in the other benchmarks, the persistent strategy achieves performance that closely approaches the minimal-overhead direct-map strategy for the "WC" trace. Finally, the table shows that though the shared-mapping strategy benefits from better reuse characteristics and achieves better performance with the `sendfile()`-based workload, the software-based strategy still performs better than both the shared or single-use IOMMU strategies for all workloads.

8.4 Sources of Reuse

The benchmarks explored in this study show varying levels of reuse, as enumerated in the "Reuse" column of Tables 3, 4, 5, and 6. For these network-based workloads, there are two primary sources of reuse: reuse within the network-buffer (`skbuff`) allocator of the operating system, and reuse among the payload buffers provided from user-space for `sendfile()` operations. For each of these types of reuse, there is both spatial and temporal reuse.

Network buffers (called `skbuffs` in Linux) are data buffers managed and allocated by the operating system to either hold the data copied in as payload from a trans-

mitting application, to hold the packet header that is to be prepended onto a zero-copy data payload packet, or to hold data that is received from the network interface. Spatial reuse of an IOMMU mapping happens when more than one usable `skbuff` can be allocated out of a single memory page, since pages are the granularity of IOMMU mappings. For transmitted packet data, this spatial reuse happens when either the packet size (which may be dictated by the maximum transmission unit (MTU) size) is less than that of a physical page, or when the network stack and network card are not utilizing TCP segmentation offloading (TSO). For packet headers prepended onto zero-copy `sendfile()` packets, spatial reuse can be fairly common because many of the small TCP/IP headers can be allocated from a single physical page.

Spatial reuse of `skbuffs` is entirely dependent on the behavior of the `skbuff` allocator, however. As is illustrated by the lack of reuse in the “Shared” IOMMU mappings for received buffers, when the PV-Linux `skbuff` allocator dedicates an entire physical page to a single network buffer, spatial reuse is completely eliminated. Likewise, temporal reuse of `skbuffs` is dependent on the behavior of the `skbuff` allocator. Clearly the default `skbuff` allocator behavior is enabling some temporal reuse, because received packets benefit from IOMMU-mapping reuse in the “Persistent” case (and as previously established, spatial reuse for this receive case is not possible given the allocator’s design).

Similarly, for payload data transmitted via zero-copy `sendfile`, spatial reuse is dependent on the packet size, the page size, and the presence (or lack) of TSO capability. Temporal reuse, however, is dependent on the application reuse patterns. Kim *et al.* have explored NIC-based data caching using the same web workloads that are presented in Table 6 and found significant opportunity for reuse [13], so temporal reuse of IOMMU mappings for these payload buffers is expected.

The experimental prototype hardware used in this prototype does not support MTU sizes larger than 1500 bytes and does not support TSO, and so some additional spatial reuse is present in these experiments that might not be present on different hardware. The “Shared” and “Persistent” strategies effectively recapture part of the IOMMU-mapping efficiency that might be gained simply by using TSO- or large-MTU-capable hardware with large messages. In both cases (IOMMU-mapping reuse or large-packet aggregation) a single IOMMU mapping is used for all the data within a physical page. However, the “Shared” and “Persistent” strategies leverage the abundant spatial reuse for workloads that also have many small packets, as in the VoIP benchmark and several of the web workloads.

9 Related Work

Contemporary commodity virtualization solutions forbid direct I/O access and instead use software to implement both protection and sharing of I/O resources among untrusted guest operating systems. Confining direct I/O accesses only within the trusted VMM ensures that all DMA descriptors used by hardware have been constructed by trusted software. Though commodity VMMs confine direct I/O within privileged software, they provide shared access to their unprivileged VMs using different software interfaces. For example, the Denali isolation kernel provides a high-level interface that operates on packets [22]. The Xen VMM provides an interface that mimics that of a real network interface card but abstracts away many of the register-level management details [9]. VMware can support either an emulated register-level interface that implements the precise semantics of a hardware NIC, or it can support a higher-level interface similar to Xen’s [19, 21].

IBM’s high-availability virtualization platforms feature IOMMUs and can support direct I/O by untrusted guest operating systems. The POWER4 platform supports logical partitioning of hardware resources among guest operating systems but does not permit concurrent sharing of resources [12]. The POWER5 platform adds support for concurrent sharing using software, effectively sacrificing direct I/O access to gain sharing [4]. This sharing mechanism works similarly to commodity solutions, effectively confining direct I/O access within what IBM refers to as a “Virtual I/O Server”. Unlike commodity VMMs, however, this software-based interface is used solely to gain flexibility, not safety. When a device is privately assigned to a single untrusted guest OS, the POWER5 platforms can still use its IOMMU to support safe, direct I/O access.

Previous studies have found that software-based approaches for I/O sharing and protection are especially costly for network I/O. Sugerman *et al.* reported a factor-of-6 network overhead penalty compared to native OS execution in a 2001 study of VMware’s network virtualization [19]. Menon *et al.* later reached similar results using the Xen virtual machine monitor, reporting a factor-of-5 penalty versus native execution in 2005 [16]. Menon *et al.* subsequently developed software-based mechanisms to reduce this overhead for transmit-oriented workloads but did not find any such mechanism for receive-based workloads [15].

The high overhead of software-based network virtualization motivated recent research toward hardware-based techniques that support simultaneous, direct-access network I/O by untrusted guest operating systems. Liu *et al.* developed an Infiniband-based prototype that supports direct access by applications running within un-

trusted virtualized guest operating systems [14]. This work adopted the Infiniband model of registration-based direct I/O memory protection, in which trusted software (the VMM) must validate and register the application's memory buffers before those buffers can be used for network I/O. Registration is similar to programming an IOMMU but has different overhead characteristics, because registrations require interaction with the device rather than modification of IOMMU page table entries. Furthermore, unlike an IOMMU, registration alone cannot provide any protection against a malfunctioning device, since the protection mechanism is partially enforced within the I/O device.

Willmann *et al.* previously developed an Ethernet-based prototype that also supports concurrent, direct network access by untrusted guest operating systems [23]. Rather than relying on hardware-based buffer registration for I/O protection, that work introduced the software-based mechanism for ensuring DMA transaction safety that is described in Section 4 of this paper. This method is based on validating DMA descriptors to be enqueued to a device and on guaranteeing the integrity of those descriptors throughout the duration of an I/O transaction. Like registration-based virtualized hardware, this software-based strategy offers no protection against faulty device behavior.

Raj and Schwan also developed an Ethernet-based prototype device that supports shared, direct I/O access by untrusted guests [17]. Because of hardware-implementation constraints, their prototype has limited addressability of main memory and thus requires all network data to be copied through VMM-managed bounce-buffers. This strategy permits the VMM to validate each buffer but does not provide any protection against faulty accesses by the device within its addressable memory range.

AMD and Intel have recently proposed the addition of IOMMUs to their upcoming architectures [3, 11]. However, IOMMUs are an established component in high-availability server architectures [6]. Ben-Yehuda *et al.* recently explored the TCP-stream network performance of IBM's state-of-the-art IOMMU-based architectures using both non-virtualized, "bare-metal" Linux and paravirtualized Linux running under Xen [7]. As is found in this paper, they reported that the state-of-the-art single-use IOMMU-management strategy can incur significant overhead. They also identified platform-specific architectural limitations that reduce performance, such as the inability to individually replace IOMMU mappings without globally flushing the CPU cache. They hypothesized that modifications to the single-use IOMMU-management strategy could avoid such penalties. Though the GART-based IOMMU implementation used in this paper does not incur the cache-

flush penalties associated with the IBM platform, single-use mappings are costly nonetheless. Furthermore, this paper proposes two specific strategies for IOMMU management that reduce IOMMU-related overhead, and examines their safety characteristics and effectiveness at reducing overhead to increase performance across a variety of real-world workloads.

10 Conclusions

This paper has evaluated a variety of DMA memory protection strategies for direct access to I/O devices within virtual machine monitors by untrusted guest operating systems. All of these strategies prevent the guest operating systems from directing the device to access memory that does not belong to that guest. The strategies do, however, differ in their performance overhead, the level of intra-guest protection, and their ability to deal with misbehaving devices.

The traditional single-use strategy provides inter-guest protection at the greatest cost, consuming from 6–26% of the CPU. However, there is significant opportunity to reuse IOMMU mappings, which in turn can reduce the cost of providing protection. This reuse and its efficiency advantages are demonstrated by the new shared- and persistent-mapping strategies introduced in this paper. Multiple concurrent network transmit operations are typically able to share the same mappings 20–40% of the time, yielding small performance improvements. However, due to Xen's I/O architecture, network receive operations are usually unable to share mappings. In contrast, using persistent mappings with a limit of 131,072 mappings enables nearly 100% reuse in almost all cases, reducing the overhead of protection to only 2–13% of the CPU.

The protection strategy supported by the VMM and the OS can also greatly affect the degree to which each guest OS can potentially protect itself by isolating the behavior of the hardware or isolating its own device drivers. Though the direct-map strategy has the least overhead, it is the only strategy that provides no mechanism for the guest OS to protect itself from its own device drivers. The persistent-mapping strategy, however, offers nearly the same performance as the direct-map strategy while still allowing some protection against misbehaving device drivers.

Finally, the software-based protection strategy evaluated in this paper performs better than two of the IOMMU-based strategies (single-use and shared), consuming only 3–15% of the CPU for protection. However, the software-based mechanism still maintains strict inter-guest memory protection. And though it cannot guard against errors that originate in the hardware, the software-based strategy supports the implementation of

enhanced intra-guest driver isolation. Therefore, an IOMMU-based protection strategy does not necessarily deliver superior performance or protection relative to software-only strategies.

Acknowledgments

We wish to thank Muli Ben-Yehuda and Ben-Ami Yasour for contributing IOMMU benchmark data for the IBM Calgary and Intel VT-d IOMMU platforms. We also wish to thank this paper's conference shepherd, Michael Swift, and our anonymous reviewers for their insightful comments and suggestions that improved this paper.

References

- [1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Oct. 2006).
- [2] ADVANCED MICRO DEVICES. *Secure Virtual Machine Architecture Reference Manual*, May 2005. Revision 3.01.
- [3] ADVANCED MICRO DEVICES. *AMD I/O Virtualization Technology (IOMMU) Specification*, Feb. 2007. Publication 34434, Revision 1.20.
- [4] ARMSTRONG, W. J., ARNDT, R. L., BOUTCHER, D. C., KOVACS, R. G., LARSON, D., LUCKE, K. A., NAYAR, N., AND SWANBERG, R. C. Advanced virtualization capabilities of POWER5 systems. *IBM Journal of Research and Development* 49, 4/5 (2005), 523–532.
- [5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (Oct. 2003).
- [6] BEN-YEHUDA, M., MASON, J., KRIEGER, O., XENIDIS, J., DOORN, L. V., MALLICK, A., NAKAJIMA, J., AND WAHLIG, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *Proceedings of the Linux Symposium* (July 2006).
- [7] BEN-YEHUDA, M., XENIDIS, J., OSTROWSKI, M., RISTER, K., BRUEMMER, A., AND DOORN, L. V. The price of safety: Evaluating IOMMU performance. In *Proceedings of the 2007 Linux Symposium* (July 2007).
- [8] DEVINE, S., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent #6,397,242* (Oct. 1998).
- [9] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the On Demand IT Infrastructure (OASIS)* (Oct. 2004).
- [10] INTEL. *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*, Apr. 2005. Revision 2.0.
- [11] INTEL CORPORATION. *Intel Virtualization Technology for Directed I/O*, May 2007. Order Number D51397-002, Revision 1.0.
- [12] JANN, J., BROWNING, L. M., AND BURUGULA, R. S. Dynamic reconfiguration: Basic building blocks for autonomic computing on ibm pseries servers. *IBM Systems Journal* 42, 1 (2003), 29–37.
- [13] KIM, H., PAI, V. S., AND RIXNER, S. Improving web server throughput with network interface data caching. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 2002), pp. 239–250.
- [14] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (June 2006).
- [15] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in Xen. In *Proceedings of the USENIX Annual Technical Conference* (June 2006).
- [16] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the ACM/USENIX Conference on Virtual Execution Environments* (June 2005).
- [17] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing* (June 2007).
- [18] SHAFER, J., AND RIXNER, S. RiceNIC: A reconfigurable network interface for experimental research and education. In *Proceedings of the Workshop on Experimental Computer Science* (June 2007).
- [19] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).
- [20] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems* 23, 1 (Feb. 2005), 77–110.
- [21] VMWARE INC. VMware ESX server: Platform for virtualizing servers, storage and networking. http://www.vmware.com/pdf/esx_datasheet.pdf, 2006.
- [22] WHITAKER, A., SHAW, M., AND GRIBBLE, S. Scale and performance in the Denali isolation kernel. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2002).
- [23] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture* (Feb. 2007).

Bridging the Gap between Software and Hardware Techniques for I/O Virtualization

Jose Renato Santos[‡]

Yoshio Turner[‡]

G.(John) Janakiraman^{‡*}

Ian Pratt[§]

[‡]*Hewlett Packard Laboratories, Palo Alto, CA*

[§]*University of Cambridge, Cambridge, UK*

Abstract

The paravirtualized I/O driver domain model, used in Xen, provides several advantages including device driver isolation in a safe execution environment, support for guest VM transparent services including live migration, and hardware independence for guests. However, these advantages currently come at the cost of high CPU overhead which can lead to low throughput for high bandwidth links such as 10 gigabit Ethernet. Direct I/O has been proposed as the solution to this performance problem but at the cost of removing the benefits of the driver domain model. In this paper we show how to significantly narrow the performance gap by improving the performance of the driver domain model. In particular, we reduce execution costs for conventional NICs by 56% on the receive path, and we achieve close to direct I/O performance for network devices supporting multiple hardware receive queues. These results make the Xen driver domain model an attractive solution for I/O virtualization for a wider range of scenarios.

1 Introduction

In virtual machine environments like VMware [12], Xen [7], and KVM [23], a major source of performance degradation is the cost of virtualizing I/O devices to allow multiple guest VMs to securely share a single device. While the techniques used for virtualizing CPU and memory resources have very low overhead leading to near native performance [29][7][4], it is challenging to efficiently virtualize most current I/O devices. Each interaction between a guest OS and an I/O device needs to undergo costly interception and validation by the virtualization layer for security isolation and for data multiplexing and demultiplexing [28]. This problem is particularly acute when virtualizing high-bandwidth network interface devices because frequent software interactions with

the device are needed to handle the high rate of packet arrivals.

Paravirtualization [30] has been proposed and used (e.g., in Xen [7][13]) to significantly shrink the cost and complexity of I/O device virtualization compared to using full device emulation. In this approach, the guest OS executes a paravirtualized (PV) driver that operates on a simplified abstract device model exported to the guest. The real device driver that actually accesses the hardware can reside in the hypervisor, or in a separate device driver domain which has privileged access to the device hardware. Using device driver domains is attractive because they allow the use of legacy OS device drivers for portability, and because they provide a safe execution environment isolated from the hypervisor and other guest VMs [16][13]. Even with the use of PV drivers, there remains very high CPU overhead (e.g., factor of four) compared to running in non-virtualized environments [18][17], leading to throughput degradation for high bandwidth links (e.g., 10 gigabits/second Ethernet). While there has been significant recent progress making the transmit path more efficient for paravirtualized I/O [17], little has been done to streamline the receive path, the focus of this paper.

To avoid the high performance overheads of software-based I/O device virtualization, efforts in academia and industry are working on adding, to varying degrees, hardware support for virtualization into I/O devices and platforms [10][24][32][22][3][6]. These approaches present a tradeoff between efficiency and transparency of I/O device virtualization. In particular, using hardware support for “direct I/O” [32][24][22], in which a device presents multiple logical interfaces which can be securely accessed by guest VMs bypassing the virtualization layer, results in the best possible performance, with CPU cost close to native performance. However, direct I/O lacks key advantages of a dedicated driver domain model: device driver isolation in a safe execution environment avoiding guest domain corruption by buggy drivers, etc.,

*Currently at Skytap.

and full support for guest VM transparent services including live migration [27] [21] [11] and traffic monitoring. Restoring these services would require either additional support in the devices or breaking the transparency of the services to the guest VM. In addition, it is difficult to exploit direct I/O in emerging virtual appliance models of software distribution which rely on the ability to execute on arbitrary hardware platforms. To use direct I/O, virtual appliances would have to include device drivers for a large variety of devices increasing their complexity, size, and maintainability.

In this paper, we significantly bridge the performance gap between the driver domain model and direct I/O in the Xen virtual machine environment, making the driver domain model a competitive and attractive approach in a wider range of scenarios. We first present a detailed analysis of the CPU costs of I/O operations for devices without hardware support for virtualization. Based on this analysis we present implementation and configuration optimizations, and we propose changes to the software architecture to reduce the remaining large costs identified by the analysis: per-packet overheads for memory management and protection, and per-byte data copy overheads. Our experimental results show that the proposed modifications reduce the CPU cost by 56% for a streaming receive microbenchmark. In addition to improving the virtualization of conventional network devices, we propose extensions to the Xen driver domain model to take advantage of emerging network interface devices that provide multiple hardware receive queues with packet demultiplexing performed by the device based on packet MAC address and VLAN ID [10]. The combination of multi-queue devices with our software architecture extensions provides a solution that retains all the advantages of the driver domain model and preserves all the benefits of virtualization including guest-transparent migration and other services. Our results show that this approach has low overhead, incurring CPU cost close to that of direct I/O for a streaming receive microbenchmark.

The rest of the paper is organized as follows. Section 2 reviews the Xen driver domain model. Section 3 presents a detailed analysis and breakdown of the costs of I/O virtualization. Section 4 presents our implementation and architectural changes that significantly improve performance for the driver domain model. Section 5 discusses how configuration settings affect I/O virtualization performance, and Section 6 presents our conclusions.

2 Xen Network I/O Architecture

Figure 1 shows the architecture of Xen paravirtualized (PV) networking. Guest domains (i.e., running virtual machines) host a paravirtualized device driver, netfront,

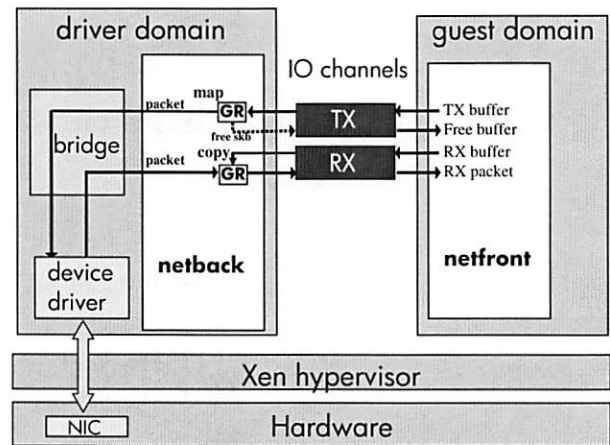


Figure 1: Xen PV driver architecture

which interacts with the device indirectly through a separate device driver domain, which has privileged access to the hardware. Driver domains directly access hardware devices that they own; however, interrupts from these devices are first handled by the hypervisor which then notifies the corresponding driver domain through virtual interrupts. Netfront communicates with a counterpart backend driver called netback in the driver domain, using shared memory I/O channels. The driver domain uses a software bridge to route packets among the physical device and multiple guests through their netback interfaces.

Each I/O channel comprises of an event notification mechanism and a bidirection ring of asynchronous requests carrying I/O buffer descriptors from netfront to netback and the corresponding responses. The event notification mechanism enables netfront and netback to trigger a virtual interrupt on the other domain to indicate new requests or responses have been posted. To enable driver domains to access I/O buffers in guest memory Xen provides a page grant mechanism. A guest creates a grant reference providing access to a page and forwards the reference as part of the I/O buffer descriptor. By invoking a hypercall, the driver domain uses the grant reference to access the guest page. For transmit (TX) requests the driver domain uses a hypercall to map the guest page into its address space before sending the request through the bridge. When the physical device driver frees the page a callback function is automatically invoked to return a response to netfront which then revokes the grant. For RX requests netfront posts I/O buffer page grants to the RX I/O channel. When netback receives a packet from the bridge it retrieves a posted grant from the I/O channel and issues a grant copy hypercall to copy the packet to the guest page. Finally, netback sends a response to the guest via the RX channel indicating a packet is available.

Table 1: Classes grouping Linux functions

Class	Description
driver	network device driver and netfront
network	general network functions
bridge	network bridge
netfilter	network filter
netback	netback
mem	memory management
interrupt	interrupt, softirq, & Xen events
schedule	process scheduling & idle loop
syscall	system call
time	time functions
dma	dma interface
hypercall	call into Xen
grant	issuing & revoking grant

Table 2: Classes grouping Xen Function

Class	Description
grant	grant map unmap or copy operation
schedule	domain scheduling
hypercall	hypercall handling
time	time functions
event	Xen events
mem	memory
interrupt	interrupt
entry	enter/exit Xen (hypercall, interrupt, fault),
traps	fault handling (also system call intercept)

3 Xen Networking Performance Analysis

This section presents a detailed performance analysis of Xen network I/O virtualization. Our analysis focuses on the receive path, which has higher virtualization overhead than the transmit path and has received less attention in the literature. We quantify the cost of processing network packets in Xen and the distribution of cost among the various components of the system software. Our analysis compares the Xen driver domain model, the Xen direct I/O model, and native Linux. The analysis provides insight into the main sources of I/O virtualization overhead and guides the design changes and optimizations we present in Section 4.

3.1 Experimental Setup

We run our experiments on two HP c-class blade servers BL460c connected through a gigabit Cisco Catalyst Blade Switch 3020. Each server has two 3GHz Intel Xeon 5160 CPUs (two dual-core CPUs) with 4MB of L2 cache each, 8GB of memory, and two Broadcom NetXtreme II BCM57085 gigabit Ethernet Network Interface Cards (NICs). Although each server had two NICs, only one was used in each experiment presented in this paper.

Table 3: Global function grouping

Class	Description
xen0	Xen functions in domain 0 context
kernel0	kernel functions in domain 0
grantcopy	data copy in grant code
xen	Xen functions in guest context
kernel	kernel functions in guest
usercopy	copy from kernel to user buffer

To generate network traffic we used the netperf[1] UDP_STREAM microbenchmark. It would be difficult to separate the CPU costs on the transmit and receive paths using TCP, which generates ACK packets in the opposite direction of data packets. Therefore, we used unidirectional UDP instead of TCP traffic. Although all results are based on UDP traffic, we expect TCP to have similar behavior at the I/O virtualization level.

We used a recent Xen unstable¹ distribution with paravirtualized Linux domains (i.e. a modified Linux kernel that does not require CPU virtualization support) using linux-2.6.18-xen². The system was configured with one guest domain and one privileged domain 0 which was also used as driver domain. For direct I/O evaluation the application was executed directly in domain 0. Both domain 0 and the guest domain were configured with 512MB of memory and a single virtual CPU each. The virtual CPUs of the guest and domain 0 were pinned to different cores of different CPU sockets³.

We use OProfile [2][18] to determine the number of CPU cycles used in each Linux and Xen function when processing network packets. Given the large number of kernel and hypervisor functions we group them into a small number of classes based on their high level purpose as described in Tables 1 and 2. In addition, when presenting overall results we group the functions in global classes as described in Table 3.

To provide safe direct I/O access to guests an IOMMU [8] is required to prevent device DMA operations from accessing other guests' memory. However, we were not able to obtain a server with IOMMU support. Thus, our results for direct I/O are optimistic since they do not include IOMMU overheads. Evaluations of IOMMU overheads are provided in [9][31].

3.2 Overall Cost of I/O Virtualization

Figure 2 compares the CPU cycles consumed for processing received UDP packets in Linux, and in Xen with direct I/O access from the guest and with Xen paravirtualized (PV) driver.

The graph shows results for three different sizes of UDP messages⁴: 52, 1500 and 48000 bytes. A message with 52 bytes corresponds to a typical TCP ACK,

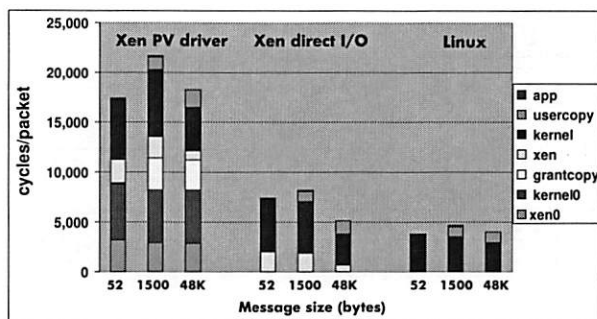


Figure 2: CPU usage to receive UDP packets

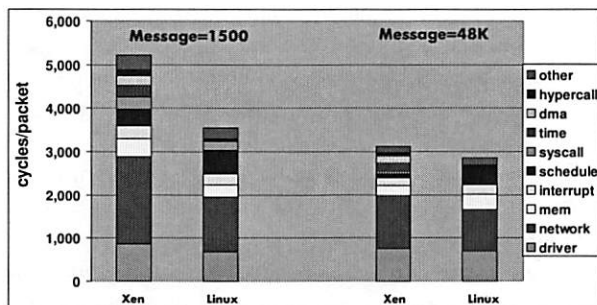


Figure 3: Kernel CPU cost for direct I/O

while a message with 1500 bytes corresponds to the maximum ethernet packet size. Using messages with 48000 bytes also generates maximum size packets but reduces the overhead of system calls at the kernel and application interface by delivering data in larger chunks. The experiments with maximum size packets in Figure 2 and in the remainder results presented in this paper were able to saturate the gigabit link. Experiments with 52-byte packets were not able to saturate the network link as the receive CPU becomes the bottleneck. To avoid having an overloaded CPU with a high number of dropped packets, we throttled the sender rate for small packets to have the same packet rate as with the large packet sizes.

The results show that in the current Xen implementation, the PV driver model consumes significantly more CPU cycles to process received packets than Linux. Direct I/O has much better performance than the PV driver, but it still has significant overhead when compared to Linux, especially for small message sizes. In the rest of this paper we analyze the sources of I/O virtualization overheads in detail, and based on this analysis we propose several changes in the design and implementation of network I/O virtualization in Xen.

We start by looking at the overheads when running guests with direct I/O access. It is surprising that for small message sizes Xen with direct I/O uses twice the number of CPU cycles to process received packets com-

pared to non-virtualized Linux. This is a consequence of memory protection limitations of the 64-bit x86 architecture that are not present in the 32-bit X86 architecture. The 64-bit x86 architecture does not support memory segmentation, which is used for protecting Xen memory in the 32-bit architecture. To overcome this limitation Xen uses different page tables for kernel and user level memory and needs to intercept every system call to switch between the two page tables. In our results, the overhead of intercepting system calls is negligible when using large message sizes (48000 bytes), since each system call consumes data from many received packets (32 packets with 1500 bytes). For more details on system call overheads the reader is referred to an extended version of this paper[26]. System call interception is an artifact of current hardware limitations which will be eliminated over time as CPU hardware support for virtualization [20][5] improves. Therefore, we ignore its effect in the rest of this paper and discuss only results for large message sizes (48000 bytes).

3.3 Analysis of Direct I/O Performance

Ignoring system call interception we observe that the kernel CPU cost for Xen with direct I/O is similar to Linux, as illustrated in the large message results in Figure 3. Xen with direct I/O consumes approximately 900 more CPU cycles per packet than native Linux (i.e. 31% overhead) for the receive path.

Of these 900 cycles, 250 cycles are due to paravirtualization changes in the kernel code. In particular, direct I/O in Xen has more CPU cycles compared to Linux in DMA functions. This is due to the use of a different implementation of the Linux DMA interface. The DMA interface is a kernel service used by device drivers to translate a virtual address to a bus address used in device DMA operations. Native Linux uses the default (*pci-nommu.c*) implementation, which simply returns the physical memory address associated with the virtual address. Para-virtualized Linux uses the software emulated I/O TLB code (*swiotlb.c*), which implements the Linux bounce buffer mechanism. This is needed in Xen because guest I/O buffers spanning multiple pages may not be contiguous in physical memory. The I/O bounce buffer mechanism uses an intermediary contiguous buffer and copies the data to/from its original memory location after/before the DMA operation, in case the original buffer is not contiguous in physical memory. However, the I/O buffers used for regular ethernet packets do not span across multiple pages and thus are not copied into a bounce buffer. The different number of observed CPU cycles is due to the different logic and extra checks needed to verify if the buffer is contiguous, and not due to an extra data copy.

Of the total 900 cycles/packet overhead for direct I/O, the hypervisor accounts for 650 cycles/packet as shown in Figure 6. Most of these cycles are in timer related functions, interrupt processing, and entering and exiting (*entry*) the hypervisor due to interrupts and hypercalls.

3.4 Analysis of PV Driver Performance

Xen PV driver consumes significantly more CPU cycles than direct I/O as shown in Figure 2. This is expected since Xen PV driver runs an additional domain to host the device driver. Extra CPU cycles are consumed both by the driver domain kernel and by the hypervisor which is now executed in the context of two domains compared to one domain with direct I/O. Additional CPU cycles are consumed to copy I/O data across domains using the Xen grant mechanism. The end result is that the CPU cost to process received network packets for Xen PV driver is approximately 18,200 cycles per packet which corresponds to 4.5 times the cost of native Linux. However, as we show in this paper, implementation and design optimizations can significantly reduce this CPU cost.

Of the total 18,200 CPU cycles consumed, approximately 1700 cycles are for copying data from kernel to user buffer (*usercopy*), 4300 for guest kernel functions (*kernel*), 900 for Xen functions in guest context (*xen*), 3000 for copying data from driver domain to guest domain (*grantcopy*), 5400 for driver domain kernel functions (*kernel0*), and 2900 for Xen functions in driver domain context (*xen0*). In contrast native Linux consumes approximately 4000 CPU cycles for each packets where 1100 cycles are used to copy data from kernel to user space and 2900 cycles are consumed in other kernel functions. In the following subsections we examine each component of the CPU cost for Xen PV driver to identify the specific causes of high overhead.

3.4.1 Copy overhead

We note in Figure 2 that both data copies (*usercopy* and *grantcopy*) in Xen PV driver consume a significantly higher number of CPU cycles than the single data copy in native Linux (*usercopy*). This is a consequence of using different memory address alignments for source and destination buffers.

Intel processor manuals [15] indicate that the processor is more efficient when copying data between memory locations that have the same 64-bit word alignment and even more efficient when they have the same cache line alignment. But packets received from the network are non aligned in order to align IP headers following the typical 14-byte Ethernet header. Netback copies the non aligned packets to the beginning of the granted page which by definition is aligned. In addition, since now

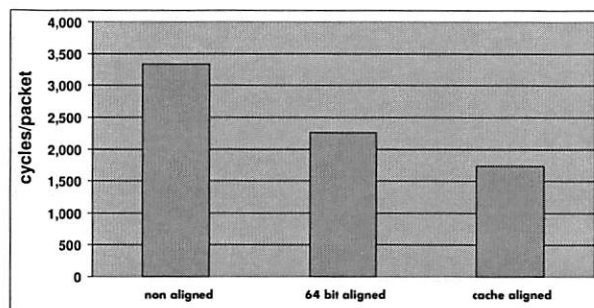


Figure 4: Alignment effect on data copy cost

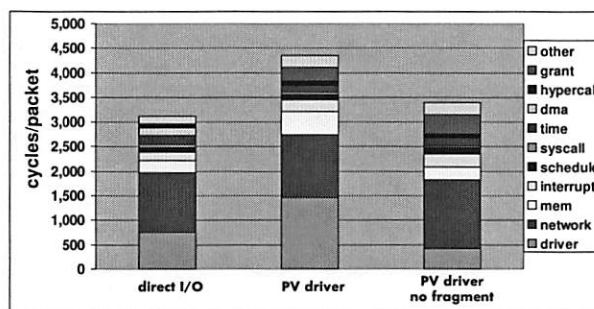


Figure 5: Kernel CPU cost

the packet starts at a 64-bit word boundary in the guest, the packet payload will start at a non word boundary in the destination buffer, due to the Ethernet header. This causes the second copy from the kernel to the user buffer in the guest to also be misaligned.

The two unaligned data copies consume significantly more CPU cycles than aligned copies. To evaluate this overhead, we modified netback to copy the packet into the guest buffer with an offset that makes the destination of the copy have the same alignment as the source. This is possible because we use a Linux guest which permits changing the socket buffer boundaries after it is allocated. Figure 4 shows the CPU cost of the grant copy for different copy alignments. The first bar shows the number of CPU cycles used to copy a 1500 byte packet when the source and destination have different word alignments. The second bar shows the number of CPU cycles when source and destination have the same 64-bit word alignment but have different cache line alignment (128 bytes), while the third bar shows the number of CPU cycles when source and destination have the same cache line alignment. These results show that proper alignment can reduce the cost of the copy by a factor of two.

3.4.2 Kernel overhead

Figure 5 compares the kernel cost for the Xen PV driver model and for the direct I/O model. Xen PV driver uses

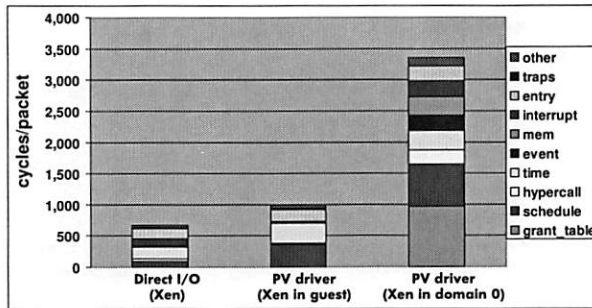


Figure 6: Hypervisor CPU cost

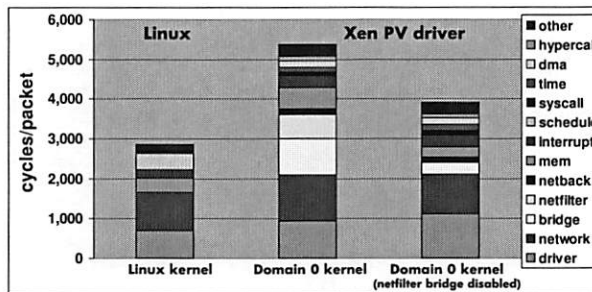


Figure 7: Driver domain CPU cost (kernel)

significantly more CPU cycles than direct I/O, especially in device driver functions. While the device drivers are different (i.e. physical device driver for direct I/O and netfront for PV driver), we would expect that the physical device driver in direct I/O would be more expensive as it has to interact with a real device, while netfront in the PV driver model only accesses the I/O channels hosted in main memory. After careful investigation we determined that the main source of increased CPU cycles in Xen PV driver is the use of fragments in guest socket buffers.

Xen supports TCP Segmentation Offloading (TSO)[17] and Large Receive Offloading (LRO)[14][19] in its virtual interfaces enabling the use of large packets spanning multiple pages for efficient packet processing. For this reason netfront posts full page buffers which are used as fragments in socket buffers, even for normal size ethernet frames. Netfront copies the first 200 bytes of the first fragment into the main linear socket buffer area, since the network stack requires the packet headers in this area. Any remaining bytes past the 200 bytes are kept in the fragment. Use of fragments thereby introduces copy overheads and socket buffer memory allocation overheads.

To measure the cost of using fragments we modified netfront to avoid using fragments. This was accomplished by pre-allocating socket buffers and posting those buffers directly into the I/O channel instead of posting fragment pages. This modification assumes packets

will not use multiple pages and thus will not require fragments, since the posted buffers are regular socket buffers and not full page fragments. Since our NIC does not support LRO [14] and it is not configured with jumbo packets, all packets received from the external network use single-page buffers in our experiments. Of course this modification cannot be used for guest to guest communication since a guest can always send large fragmented packets. The third bar in Figure 5 shows the performance results using the modified netfront. The result shows that using socket buffers with fragments is responsible for most of the additional kernel cycles for the PV driver case when compared to direct I/O.

Without the overhead of fragments the kernel CPU cost for PV driver is very close to that of direct I/O, except for small variations in the distribution of cycles among the different kernel functions. In particular, netfront in Xen PV driver now has lower cost than the physical device driver in direct I/O, as expected. Also since netfront does not access a physical device it does not need to use the software I/O TLB implementation of the DMA interface used by direct I/O. On the other hand PV drivers have the cost of issuing and revoking grants, which are not used with direct I/O.

Most of this grant cost is due to the use of expensive atomic compare and swap instructions to revoke grant privileges in the guest grant table. This is necessary because Xen uses different bits of the same grant table field to store the status of the grant (grant in use or not by the driver domain) updated by Xen, and the grant access permission (enable or revoke grant access) updated by the issuing domain. The guest must revoke and check that the grant is no longer in use by the driver domain using an atomic operation, to ensure that a driver domain does not race with grant revoking and keep an undesired reference to the page after the grant is revoked. The use of atomic compare and swap instructions to revoke a grant adds a significant number of CPU cycles in the guest kernel cost. If however these two different grant bits are stored in different words, we can ensure atomicity using less expensive operations such as memory barriers. The result with implementation optimizations presented later in Section 4 include this and other grant optimizations discussed in the following Section 3.4.3.

3.4.3 Hypervisor overhead

I/O processing consumes CPU cycles executing hypervisor functions in addition to guest kernel code. Figure 6 shows the CPU cost in Xen hypervisor functions for receiving network packets with PV driver. The graph shows the CPU cost when executing Xen functions in the context of the guest and in the context of the driver domain (i.e. domain 0), and compares them with the CPU

cost for direct I/O.

The graph shows that most of the CPU cost for the hypervisor is due to code executing in the context of the driver domain, and the larger cost components are due to grant operations and schedule functions. The hypervisor CPU cost in guest context for PV drivers is similar to the hypervisor cost for direct I/O except for a higher cost in schedule functions.

The higher cost in scheduling functions for PV driver is due to increased cache misses when accessing Xen data structures for domain scheduling purposes. With PV driver, the guest and the driver domain run on different CPUs, causing domain data structures used by scheduling functions to bounce between the two CPUs.

We identified the most expensive operations for executing grant code by selectively removing code from the grant functions. Basically they are the following: **1)** acquiring and releasing spin locks, **2)** pinning pages, and **3)** use of expensive atomic swap operations to update grant status as previously described.

All of these operations can be optimized. The number of spinlock operations can be significantly reduced by combining the operations of multiple grants in a single critical section. The use of atomic swap operations can be avoided by separating grant fields in different words as described in section 3.4.2. Note that this optimization reduces overhead in Xen and in the guest, since both of them have to access the same grant field atomically.

The cost of pinning pages can also be optimized when using grants for data copy. During a grant copy operation, the hypervisor creates temporary mappings into hypervisor address space for both source and destination of the copy. The hypervisor also pins (i.e. increment a reference counter) both pages to prevent the pages from being freed while the grant is active. However, usually one of the pages is already pinned and mapped in the address space of the current domain which issued the grant operation hypercall. Thus we can avoid mapping and pinning the domain local page and just pin the foreign page referred by the grant. It turns out that pinning pages for writing in Xen is significantly more expensive than pinning pages for read, as it requires to increment an additional reference counter using an expensive atomic instruction. Therefore, this optimization has higher performance impact when the grant is used to copy data from a granted page to a local page (as we propose below in Section 4.1) instead of the other way around.

3.4.4 Driver domain overhead

Figure 7 shows CPU cycles consumed by the driver domain kernel (2nd bar graph; domain 0) and compares it with the kernel cost in native Linux (1st bar graph). The results show that the kernel cost in the driver domain is

almost twice the cost of the Linux kernel. This is somewhat surprising, since in the driver domain the packet is only processed by the lower level of the network stack (ethernet), although it is handled by two device drivers: native device driver and netback. We observe that a large number of the CPU cycles in driver domain kernel are due to bridge and netfilter functions. Note that although the kernel has netfilter support enabled, no netfilter rule or filter is used in our experiments. The cost shown in the graph is the cost of netfilter hooks in the bridge code that are always executed to test if a filter needs to be applied. The third bar graph in the figure shows the performance of a driver domain when the kernel is compiled with the bridge netfilter disabled. The results show that most of the bridge cost and all netfilter cost can be eliminated if the kernel is configured appropriately when netfilter rules are not needed.

4 Xen Network Design Changes

In the previous section we identified several sources of overhead for Xen PV network drivers. In this section we propose architectural changes to the Xen PV driver model that significantly improve performance. These changes modify the behavior of netfront and netback, and the I/O channel protocol.

Some inefficiencies identified in Section 3 can be reduced through implementation optimizations that do not constitute architectural or protocol changes. Although some of the implementation optimizations are easier to implement in the new architecture, we evaluated their performance impact in the current architecture in order to separate their performance benefits from that of the architectural changes. The implementation optimizations include: disabling the netfilter bridge, using aligned data copies, avoiding socket buffer fragments and the various grant optimizations discussed in Sections 3.4.2 and 3.4.3.

The second bar in Figure 8 shows the cumulative performance impact of all these implementation optimizations and is used as a reference point for the performance improvements of the architectural changes presented in this section. In summary, the implementation optimizations reduce the CPU cost of Xen PV driver by approximately 4950 CPU cycles per packet. Of these, 1550 cycles are due to disabling bridge netfilter, 1850 cycles due to using cache aligned data copies, 900 cycles due to avoiding socket buffer fragments and 650 cycles due to grant optimizations.

4.1 Move Data Copy to Guest

As described in Section 3 a primary source of overhead for Xen PV driver is the additional data copy between driver domain and guest. In native Linux there is only

one data copy, as the received packet is placed directly into a kernel socket buffer by the NIC and later copied from there to the application buffer. In Xen PV driver model there are two data copies, as the received packet is first placed in kernel memory of the driver domain by the NIC, and then it is copied to kernel memory of the guest before it can be delivered to the application.

This extra cost could be avoided if we could transfer the ownership of the page containing the received packet from the driver domain to the guest, instead of copying the data. In fact this was the original approach used in previous versions of Xen [13]. The first versions of Xen PV network driver used a page flipping mechanism which swapped the page containing the received packet with a free guest page, avoiding the data copy. The original page flip mechanism was replaced by the data copy mechanism in later versions of Xen for performance reasons. The cost of mapping and unmapping pages in both guest and driver domain was equivalent to the copy cost for large 1500 byte packets, which means that page flipping was less efficient than copy for small packet sizes [25]. In addition, the page flip mechanism increases memory fragmentation and prevents the use of super-page mappings with Xen. Menon et al. [17] have shown that super-pages provide superior performance in Xen, making page flipping unattractive.

One problem with the current data copy mechanism is that the two data copies per packet are usually performed by different CPUs leading to poor data cache behavior. On an SMP machine, it is expected that an I/O intensive guest and the driver domain will be executing on different CPUs, especially if there is high I/O demand. The overhead introduced by the extra data copy can be reduced if both copies are performed by the same CPU and benefit from cache locality. The CPU will bring the packet to its cache during the first data copy. If the data is still present in the CPU cache during the second copy, this copy will be significantly faster using fewer CPU cycles. Of course there is no guarantee that the data will not be evicted from the cache before the second copy, which can be delayed arbitrarily depending on the application and overall system workload behavior. At high I/O rates, however, it is expected that the data will be delivered to the application as soon as it is received and will benefit from L2 cache locality.

We modified the architecture of Xen PV driver and moved the grant copy operation from the driver domain to the guest domain, improving cache locality for the second data copy. In the new design, when a packet is received netback issues a grant for the packet page to the guest and notifies the guest of the packet arrival through the I/O channel. When netfront receives the I/O channel notification, it issues a grant copy operation to copy the packet from a driver domain page to a local socket buffer,

and then delivers the packet to the kernel network stack.

Moving the grant copy to the guest has benefits beyond speeding up the second data copy. It avoids polluting the cache of the driver domain CPU with data that will not be used, and thus should improve cache behavior in the driver domain as well. It also provides better CPU usage accounting. The CPU cycles used to copy the packet will now be accounted to the guest instead of to the driver domain, increasing fairness when accounting for CPU usage in Xen scheduling. Another benefit is improved scalability for multiple guests. For high speed networks (e.g. 10GigE), the driver domain can become the bottleneck and reduce I/O throughput. Offloading some of the CPU cycles to multiple guests' CPUs avoids the driver domain from becoming a bottleneck improving I/O scalability. Finally, some implementation optimizations described earlier are easier to implement when the copy is done by the guest. For example, it is easier to avoid the extra socket buffer fragment discussed in Section 3.4.2. Moving the copy to the guest allows the guest to allocate the buffer *after* the packet is received from netback at netfront. Having knowledge of the received packet allows netfront to allocate the appropriate buffers with the right sizes and alignments. Netfront can allocate one socket buffer for the first page of each packet and additional fragment pages only if the packet spans multiple pages. For the same reason, it is also easier to make aligned data copies, as the right socket buffer alignment can be selected at buffer allocation time.

The third bar in Figure 8 shows the performance benefit of moving the grant copy from the driver domain to the guest. The cost of the data copy is reduced because of better use of the L2 cache. In addition, the number of cycles consumed by Xen in guest context (*xen*) increases while it decreases for Xen in driver domain context (*xen0*). This is because the cycles used by the grant operation are shifted from the driver domain to the guest. We observe that the cost decrease in *xen0* is higher than the cost increase in *xen* leading to an overall reduction in the CPU cost of Xen. This is because the grant optimizations described in Section 3.4.3 are more effective when the grant operations are performed by the guest, as previously discussed.

In summary, moving the copy from the driver domain to the guest reduces the CPU cost for Xen PV driver by approximately 2250 cycles per packet. Of these, 1400 cycles are due to better cache locality for guest copies (*usercopy*, *grantcopy*, and also *kernel* for faster accesses to packet headers in protocol processing), 600 cycles are due to grant optimizations being more effective (*xen* + *xen0*) and 250 cycles are due to less cache pollution in driver domain (*kernel0*).

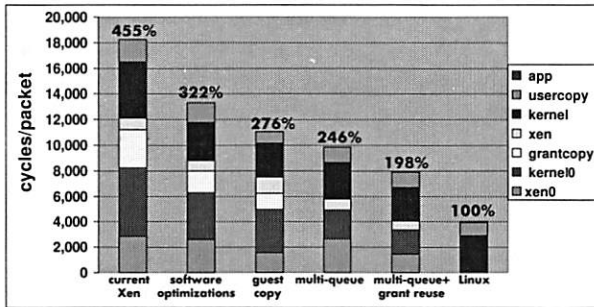


Figure 8: Xen PV driver Optimizations

4.2 Extending Grant Mechanism

Moving the grant copy to the guest requires a couple of extensions to the Xen grant mechanism. We have not yet implemented these extensions but we discuss them here.

To support copy in the receiving guest, the driver domain has to issue a grant to the memory page containing the received packet. This works fine for packets received on the physical device since they are placed in driver domain memory. In contrast, the memory buffers with packets received from other guests are not owned by the driver domain and cannot be granted to the receiving guest using the current Xen grant mechanism. Thus, this mechanism needs to be extended to provide grant transitivity allowing a domain that was granted access to another domain's page, to transfer this right to a third domain.

We must also ensure memory isolation and prevent guests from accessing packets destined to other guests. The main problem is that the same I/O buffer can be reused to receive new packets destined to different guests. When a small sized packet is received on a buffer previously used to receive a large packet for another domain, the buffer may still contain data from the old packet. Scrubbing the I/O pages after or before every I/O to remove old data would have a high overhead wiping out the benefits of moving the copy to the guest. Instead we can just extend the Xen grant copy mechanism with offset and size fields to constrain the receiving guest to access only the region of the granted page containing the received packet data.

4.3 Support for Multi-queue Devices

Although moving the data copy to the guest reduces the CPU cost, eliminating the extra copy altogether should provide even better performance. The extra copy can be avoided if the NIC can place received packets directly into the guest kernel buffer. This is only possible if the NIC can identify the destination guest for each packet and select a buffer from the respective guest's memory

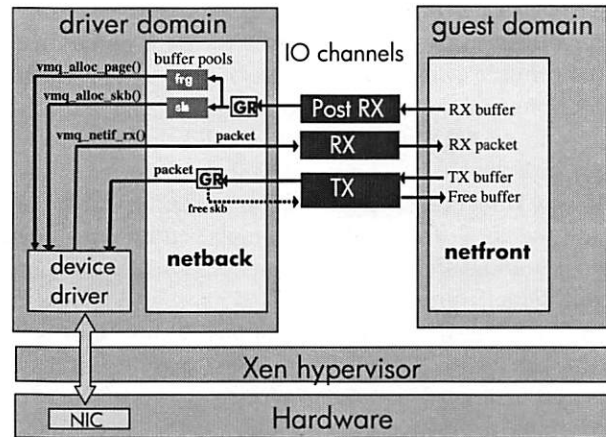


Figure 9: Multi-queue device support

to place the packet. As discussed in Section 1, NICs are now becoming available that have multiple receive (RX) queues that can be used to directly place received packets in guest memory [10]. These NICs can demultiplex incoming traffic into the multiple RX queues based on the packet destination MAC address and VLAN IDs. Each individual RX queue can be dedicated to a particular guest and programmed with the guest MAC address. If the buffer descriptors posted at each RX queue point to kernel buffers of the respective guest, the device can place incoming packets directly into guest buffers, avoiding the extra data copy.

Figure 9 illustrates how the Xen PV network driver model can be modified to support multi-queue devices. Netfront posts grants to I/O buffers for use by the multi-queue device drivers using the I/O channel. For multi-queue devices the driver domain must validate if the page belongs to the (untrusted) guest and needs to pin the page for the I/O duration to prevent the page being reassigned to the hypervisor or other guests. The grant map and unmap operations accomplish these tasks in addition to mapping the page in the driver domain. Mapping the page is needed for guest to guest traffic which traverses the driver domain network stack (bridge). Experimental results not presented here due to space limitations show that the additional cost of mapping the page is small compared to the overall cost of the grant operation.

Netfront allocates two different types of I/O buffers which are posted to the I/O channel: regular socket buffers with the right alignments required by the network stack, and full pages for use as fragments in non linear socket buffers. Posting fragment pages is optional and just needed if the device can receive large packets spanning multiple pages, either because it is configured with jumbo frames or because it supports Large Receive Offload (LRO). Netback uses these grants to map the

guest pages into driver domain address space buffers and keeps them in two different pools for each guest: one pool for each type of page. These buffers are provided to the physical device driver on demand when the driver needs to post RX descriptors to the RX queue associated with the guest. This requires that the device driver use new kernel functions to request I/O buffers from the guest memory. This can be accomplished by providing two new I/O buffer allocation functions in the driver domain kernel, *vmq_alloc_skb()* and *vmq_alloc_page()*. These are equivalent to the traditional Linux functions *netdev_alloc_skb()* and *alloc_page()* except that they take an additional parameter specifying the RX queue for which the buffer is being allocated. These functions return a buffer from the respective pool of guest I/O buffers. When a packet is received the NIC consumes one (or more in case of LRO or jumbo frame) of the posted RX buffers and places the data directly into the corresponding guest memory. The device driver is notified by the NIC that a packet arrived and then forwards the packet directly to netback. Since the NIC already demultiplexes packets, there is no need to use the bridge in the driver domain. The device driver can instead send the packet directly to netback using a new kernel function *vmq_netif_rx()*. This function is equivalent to the traditional Linux *netif_rx()* typically used by network drivers to deliver received messages, except that the new function takes an additional parameter that specifies which RX queue received the packet.

Note that these new kernel functions are not specific to Xen but enable use of multi-queue devices with any virtualization technology based on the Linux kernel, such as for example KVM[23]. These new functions only need to be used in new device drivers for modern devices that have multi-queue support. These new functions extend the current interface between Linux and network devices enabling the use of multi-queue devices for virtualization. This means that the same device driver for a multi-queue device can be used with Xen, KVM or any other Linux based virtualization technology.

We observe that the optimization that moves the copy to the guest is still useful even when using multi-queue devices. One reason is that the number of guests may exceed the number of RX queues causing some guests to share the same RX queue. In this case guests that share the same queue should still use the grant copy mechanism to copy packets from the driver domain memory to the guest. Also, grant copy is still needed to deliver local guest to guest traffic. When a guest sends a packet to another local guest the data needs to be copied from one guest memory to another, instead of being sent to the physical device. In these cases moving the copy to the guest still provides performance benefits. To support receiving packets from both the multi-queue device and

other guests, netfront receives both types of packets on the RX I/O channel shown in Figure 9. Packets from other guests arrive with copy grants that are used by netfront, whereas packets from the device use pre-posted buffers.

We have implemented a PV driver prototype with multi-queue support to evaluate its performance impact. However, we did not have a NIC with multi-queue support available in our prototype. We used instead a traditional single queue NIC to estimate the benefits of a multi-queue device. We basically modified the device driver to dedicate the single RX queue of the NIC to the guest. We also modified netback to forward guest buffers posted by netfront to the physical device driver, such that the single queue device could accurately emulate the behavior of a multi-queue device.

The fourth bar in Figure 8 shows the performance impact of using multi-queue devices. As expected the cost of grant copy is eliminated as now the packet is placed directly in guest kernel memory. Also the number of CPU cycles in *xen* for guest context is reduced since there is no grant copy operation being performed. On the other hand the driver domain has to use two grant operations per packet to map and unmap the guest page as opposed to one operation for the grant copy, increasing the number of cycles in *xen0* and reducing the benefits of removing the copy cost. However, the number of CPU cycles consumed in driver domain kernel is also reduced when using multi-queue devices. This is due to two reasons. First, packets are forwarded directly from the device driver to the guest avoiding the forwarding costs in the bridge. Second, since netback is now involved in both allocating and deallocating socket buffer structures for the driver domain, it can avoid the costs of their allocation and deallocation. Instead, netback recycles the same set of socket buffer structures in multiple I/O operations. It only has to change their memory mappings for every new I/O, using the grant mechanism to make the socket buffers point to the right physical pages containing the guest I/O buffers. Surprisingly, the simplifications in driver domain have a higher impact on the CPU cost than the elimination of the extra data copy.

In summary, direct placement of data in guest memory reduces PV driver cost by 700 CPU cycles per packet, and simplified socket buffer allocation and simpler packet routing in driver domain reduces the cost by additional 1150 cycles per packet. On the other hand the higher cost of grant mapping over the grant copy, increases the cost by 650 cycles per packet providing a net cost reduction of 1200 CPU cycles per packet. However, the benefit of multi-queue devices can actually be larger when we avoid the costs associated with grants as discussed in the next section.

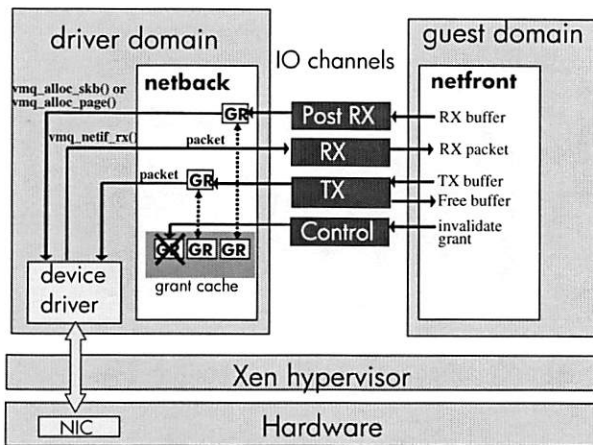


Figure 10: Caching and reusing grants

4.4 Caching and Reusing Grants

As shown in Section 3.4.3, a large fraction of Xen CPU cycles consumed during I/O operations are due to grant operation functions. In this section we describe a grant reuse mechanism that can eliminate most of this cost.

The number of grant operations performed in the driver domain can be reduced if we relax the memory isolation property slightly and allow the driver domain to keep guest I/O buffers mapped in its address space even after the I/O is completed. If the guest recycles I/O memory and reuses previously used I/O pages for new I/O operations, the cost of mapping the guest pages using the grant mechanism is amortized over multiple I/O operations. Fortunately, most operating systems tend to recycle I/O buffers. For example, the Linux slab allocator used to allocate socket buffers keeps previously used buffers in a cache which is then used to allocate new I/O buffers. In practice, keeping I/O buffer mappings for longer times does not compromise the fault isolation properties of driver domains, as the driver domain still can only access the same set of I/O pages and no pages containing any other guest data or code.

In order to reuse grants, Xen PV driver needs to be modified as illustrated in Figure 10. Netback keeps a cache of currently mapped grants for every guest. On every RX buffer posted by the guest (when using a multi-queue device) and on every TX request, netback checks if the granted page is already mapped in its address space, mapping it only if necessary. When the I/O completes, the mapping is not removed allowing it to be reused in future I/O operations. It is important, though, to enable the guest to explicitly request that a cached grant mapping be invalidated. This may be necessary, for example if the guest repurposes the page and uses it somewhere else in the guest or if it returns the page back to the hypervisor. In that case, it is desirable to revoke the grant and unmap

the granted page from the driver domain address space, in order to preserve memory isolation between driver domain and guest⁵. A new I/O control channel between netfront and netback is used for this purpose. Netfront sends grant invalidation requests and netback sends confirmation responses after the granted page is unmapped. In summary, this mechanism preserves the isolation between driver domain and guest memory (only I/O buffer pages are shared) and avoids the overhead of mapping and unmapping pages on every I/O operation.

Since the amount of memory consumed for each grant cached in netback is relatively small when compared with the page size, the maximum number of cached grants should be limited only by kernel address space available in the driver domain. The address space reserved for the kernel is significantly larger than the size of a typical active set of I/O buffers. For example, 1GB of the Linux address space is reserved for the kernel; although some of this space is used by other kernel functions, a large fraction of this space can be used for dynamic mapping of guest I/O buffer pages. The size of the active set of I/O buffers is highly dependent on the workload, but typically it should not exceed a few megabytes. In practice, the driver domain should be able to map most of the active I/O buffer pages in its address space for a large number of guests. Thus we expect that the grant reuse mechanism will provide close to a 100% hit rate in the netback grant cache, except for unlikely scenarios with more than hundreds of guests. Thus, the overhead of grant mapping can be reduced to almost zero in practical scenarios, when the guest buffer allocation mechanism promotes buffer reuse.

We have not yet implemented the complete grant reuse mechanism described above. Instead, for evaluation purposes we implemented a simplified mechanism that avoids the use of grants at all. We modified the I/O channel to use physical page addresses directly instead of grants to specify RX buffers. Netfront specifies the machine physical addresses of I/O buffers in the I/O channel requests, and the driver domain uses these addresses directly when programming the DMA operations. Note that this is not a safe mechanism since there is no validation that the physical page used for I/O belongs to the corresponding guest and no guarantee that the page is pinned. Thus, this mechanism is used here just for performance evaluation purposes. The mechanism completely avoids the use of grants and estimates the benefit of the grant reuse mechanism when the hit rate on cached grants is 100%. Although this is an optimistic estimation it is expected to accurately predict actual performance, since we expect to achieve close to a 100% hit rate on cached grants by using appropriate buffer allocation mechanisms. However, validation of this estimation is planned as future work.

The fifth bar in Figure 8 shows the performance benefit of caching and reusing grants in the driver domain. As expected most of the benefit comes from reduced number of cycles in *xen0* used for grant operations. In summary, grant reuse reduces the cost of PV driver by 1950 CPU cycles per packet, which combined with all other optimizations reduces the cost by 10300 CPU cycles. Although the optimizations described so far provide significant cost reduction, Xen I/O virtualization still has twice the cost of native I/O in Linux. However, it is possible to reduce this cost even further by properly setting system configuration parameters as described in the next section.

5 Tuning I/O Virtualization Configuration

5.1 Decoupling Driver Domain from Domain 0

Although the current architecture of the Xen PV driver model is flexible and allows the use of dedicated driver domains used exclusively for doing I/O, in practice most Xen installations are configured with domain 0 acting as the driver domain for all physical devices. The reason is that there is still no mechanism available in Xen that leverages the fault isolation properties of dedicated driver domains. For example there is currently no available tool that automatically detects driver domain faults and restarts them. Since hosting all drivers in domain 0 is simpler to configure this is typically the chosen configuration.

However, hosting all device drivers in domain 0 prevents tuning some configuration options that optimize I/O performance. Since domain 0 is a general purpose OS it needs to support all standard Linux utilities and Xen administration tools, thus requiring a full fledged kernel with support for all features of a general purpose operating system. This limits the flexibility in configuring the driver domain with optimized I/O configuration options. For example, disabling the bridge netfilter option of the kernel significantly improves performance as shown in Section 3.4.4. However, network tools such as iptables available in standard Linux distributions do not work properly if this kernel configuration option is disabled. This has prevented standard Linux vendors such as RedHat and Novell to enable this kernel option in their standard distribution, thus preventing this I/O optimization in practice. Separating the driver domain from domain 0 allows us to properly configure the driver domain with configurations that are optimized for I/O.

5.2 Reducing Interrupt Rate

The architectural changes discussed in Section 4 address two important sources of overhead in I/O virtualization:

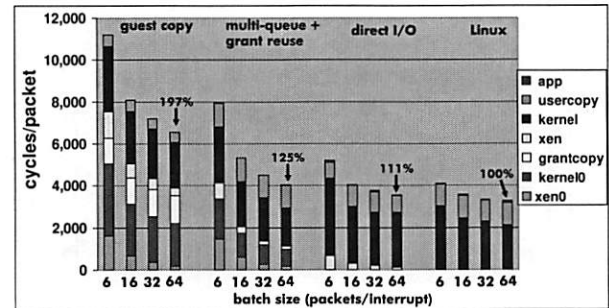


Figure 11: Interrupt throttling

extra data copies and the Xen grant mechanism. These are the most important overheads that are directly proportional to the amount of received traffic. Data copy overheads are proportional to the number of received bytes while grant overheads are proportional to the number of received packets.

Most of the remaining overheads are proportional to the number of times the driver domain and the guest are scheduled (this is different from the number of packets since both domains can process multiple packets in a single run). These additional overheads include processing physical interrupts, virtual interrupts, event delivery, domain scheduling, hypercalls, etc. Even though some of these overheads are also present in Linux, they have a higher impact when I/O is virtualized due to the additional levels of software such as the hypervisor and the driver domain. For example, a device interrupt for a received packet causes CPU cycles to be consumed both in the hypervisor interrupt handler and in the driver domain handler. Additional CPU cycles are consumed when handling the virtual interrupt in the guest after the packet is delivered through the I/O channel. Increasing the number of packets that are processed each time the guest or the driver domain is scheduled should reduce the remaining performance overheads.

On the receive path the driver domain is typically scheduled to process I/O when an interrupt is generated by the physical device. Most network devices available today can delay the generation of interrupts until multiple packets are received and thus reduce the interrupt rate at high throughputs. The interrupt rate for received packets can be controlled by special device driver parameters usually known as interrupt coalescing parameters. Interrupt coalescing parameters specify not only the number of packets per interrupt, but also a maximum delay after the last received packet. An interrupt is generated when either the specified number of packets is received or when the maximum specified delay is reached. This mechanism allows us to limit the interrupt rate at high I/O rates while preserving low latency at low I/O rates.

Configuring device coalescing parameters enables us

to change the number of packets processed in each run of the driver domain and thus amortize the overheads over a larger number of packets. All packets received in a single interrupt by the device driver are queued in netback queues before being processed. This causes netback to process packets in batches of the same size as the device driver. Since netback only notifies netfront after all packets in the batch are processed and added to the I/O channel, the device coalescing parameters also limit the virtual interrupt rate in the guest domains, and thus amortize the I/O overheads in the guest as well.

Figure 11 shows the effect of interrupt coalescing on the CPU cost of I/O virtualization. The graph shows CPU cost for four cases: 1) Optimized PV driver using a traditional network device (guest copy), 2) Optimized PV driver using a multi-queue device (multi-queue + grant reuse), 3) Direct I/O, 4) native Linux. The figure shows the CPU cost for different interrupt rates for each of the four cases. The Linux default interrupt coalescing parameters for our Broadcom NIC is 6 pkt/int (packets per interrupt) with a maximum latency of 18 μ s. While we varied the batch size from 6 to 64 pkt/int we kept the default maximum interrupt latency of 18 μ s in all results presented in Figure 11, preserving the packet latency at low throughputs.

The graph shows that the CPU cost for the PV driver is significantly reduced when the number of packets processed per interrupt is increased, while the effect is less pronounced for both Linux and Direct I/O. This result confirms that most of the remaining I/O virtualization overheads are proportional to the interrupt rate. In Linux, the default value of 6 pkt/int performs almost as well as a large batch of 64 pkt/int. This suggests that the default interrupt coalescing parameters for network device drivers that work well for native Linux, are not the best configuration for Xen PV driver.

Experimental results not shown here due to space limitations show that interrupt coalescing achieves approximately the same CPU cost reduction for the original Xen PV driver configuration as it does for our optimized PV driver without hardware multi-queue support, i.e., approximately 4600 cycles per packet for batches of size 64. This indicates that interrupt coalescing and the other optimizations presented in this paper are complementary.

In summary, the results show that software-only optimizations reduce I/O virtualization overheads for the Xen driver domain model from 355% to 97% of the Linux cost for high throughput streaming traffic. Moreover, the use of hardware support for I/O virtualization enables us to achieve close to native performance: multi-queue devices can reduce the overhead to only 25% of the Linux cost while direct I/O has 11% overhead. The main difference is due to the cost of executing netfront and netback when using Xen PV driver. For real applica-

tions, the effective cost difference between direct I/O and the driver domain model should much be lower, since applications will use CPU cycles for additional work besides I/O processing. The low cost of the driver domain model combined with multi-queue support suggest that it is an attractive solution for I/O virtualization.

6 Conclusion

The driver domain model used in Xen has several desired properties. It isolates the address space of device drivers from guest and hypervisor code preventing buggy device drivers from causing system crashes. Also, driver domains can support guest VM transparent services such as live migration and network traffic monitoring and control (e.g. firewalls).

However, the driver domain model needs to overcome the address space isolation in order to provide I/O services to guest domains. Device drivers need special mechanisms to gain access to I/O data in other guest domains and to move the I/O data bytes to and from those domains. In Xen this is accomplished through the grant mechanism. In this paper we propose several architectural changes that reduce the performance overhead associated with driver domains models. First we propose two mechanisms that reduce the cost of moving the I/O data bytes between guest and driver domains: 1) we increase the cache locality of the data copy by moving the copy operation to the receiving guest CPU and 2) we avoid the data copy between domains by using hardware support of modern NICs to place data directly into guest memory. Second we minimize the cost of granting the driver domain access to guest domain pages by slightly relaxing the memory isolation property to allow a set of I/O buffers to be shared between domains across multiple I/O operations. Although these architectural changes were done in the context of Xen they are applicable to the driver domain model in general.

Our work demonstrates that it is possible to achieve near direct I/O and native performance while preserving the advantages of a driver domain model for I/O virtualization. We advocate that the advantages of the driver domain model outweigh the small performance advantage of direct I/O in most practical scenarios.

In addition, this paper identified several low-level optimizations for the current Xen implementation which had surprisingly large impact on overall performance.

References

- [1] Netperf. www.netperf.org.
- [2] Oprofile. oprofile.sourceforge.net.
- [3] ABRAMSON, D., JACKSON, J., MUTHURASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I.,

- UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel® virtualization technology for directed I/O. *Intel® Technology Journal* 10, 3 (August 2006). www.intel.com/technology/itj/2006/v10i3/.
- [4] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS* (2006), J. P. Shen and M. Martonosi, Eds., ACM, pp. 2–13.
- [5] ADVANCED MICRO DEVICES. AMD64 architecture programmer's manual volume 2: System programming. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf, Sept 2007.
- [6] ADVANCED MICRO DEVICES, INC. IOMMU architectural specification. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf, Feb 2007. PID 34434 Rev 1.20.
- [7] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T. L., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP* (2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 164–177.
- [8] BEN-YEHUDA, M., MASON, J., KRIEGER, O., XENIDIS, J., VAN DOORN, L., MALLICK, A., NAKAJIMA, J., AND WAHLIG, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *Ottawa Linux Symposium* (2006).
- [9] BEN-YEHUDA, M., XENIDIS, J., OSTROWSKI, M., RISTER, K., BRUEMMER, A., AND VAN DOORN, L. The price of safety: evaluating IOMMU performance. In *Ottawa Linux Symposium* (2007).
- [10] CHINNI, S., AND HIREMAN, R. Virtual machine device queues. softwaredispatch.intel.com/?id=1894. Supported in Intel® 82575 gigE and 82598 10GigE controllers.
- [11] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI* (2005), USENIX.
- [12] DEVINE, S., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMware US Patent 6397242, Oct 1998.
- [13] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMS, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT Infrastructure (OASIS)* (October 2004).
- [14] GROSSMAN, L. Large Receive Offload implementation in Netetion 10GbE Ethernet driver. In *Ottawa Linux Symposium* (2005).
- [15] INTEL® CORPORATION. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. www.intel.com/products/processor/manuals/index.htm.
- [16] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI* (2004), pp. 17–30.
- [17] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in Xen. In *USENIX Annual Technical Conference* (June 2006).
- [18] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)* (June 2005).
- [19] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. In *USENIX Annual Technical Conference* (June 2008).
- [20] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel® virtualization technology: hardware support for efficient processor virtualization. *Intel® Technology Journal* 10, 3 (August 2006). www.intel.com/technology/itj/2006/v10i3/.
- [21] NELSON, M., LIM, B.-H., AND HUTCHINS, G. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference* (April 2005).
- [22] PCI SIG. I/O virtualization. www.pcisig.com/specifications/iov/.
- [23] QUMRANET. KVM: Kernel-based virtualization driver. www.qumranet.com/wp/kvm.wp.pdf.
- [24] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC* (2007).
- [25] SANTOS, J. R., JANAKIRAMAN, G., AND TURNER, Y. Network optimizations for PV guests. In *3rd Xen Summit* (Sept 2006).
- [26] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND PRATT, I. Bridging the gap between software and hardware techniques for i/o virtualization. In *HP Labs Tech Report, HPL-2008-39* (2008).
- [27] SAPUNTZAKIS, C., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *5th Symposium on Operating Systems Design and Implementation* (December 2002).
- [28] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track* (2001), Y. Park, Ed., USENIX, pp. 1–14.
- [29] WALDSPURGER, C. A. Memory resource management in VMware ESX Server. In *OSDI* (2002).
- [30] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the Denali isolation kernel. In *OSDI* (2002).
- [31] WILLMANN, P., COX, A. L., AND RIXNER, S. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference* (June 2008).
- [32] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *High Performance Computer Architecture (HPCA)* (February 2007).

Notes

¹xen-unstable.hg changeset 15521:1f348e70a5af, July 10, 2007

²linux-2.6.18-xen.hg changeset 103:a70de77dd8d3, July 10, 2007

³Using two cores in the same CPU would have improved performance due to the shared L2 cache. We chose the worst case configuration since we cannot enforce that all guests will share the same socket with the driver domain.

⁴The message size includes IP and UDP headers but does not include 14 bytes of Ethernet header per packet.

⁵Even if the guest does not request that the grant be revoked Xen will not allocate the page to another guest while the grant is active, maintaining safe memory protection between guests.

Idle Read After Write - IRAW

Alma Riska
Seagate Research
1251 Waterfront Place
Pittsburgh, PA 15222
Alma.Riska@seagate.com

Erik Riedel
Seagate Research
1251 Waterfront Place
Pittsburgh, PA 15222
Erik.Riedel@seagate.com

Abstract

Despite a low occurrence rate, silent data corruption represents a growing concern for storage systems designers. Throughout the storage hierarchy, from the file system down to the disk drives, various solutions exist to avoid, detect, and correct silent data corruption. Undetected errors during the completion of WRITES may cause silent data corruption. A portion of the WRITE errors may be detected and corrected successfully by verifying the data written on the disk with the data in the disk cache. Write verification traditionally is scheduled immediately after a WRITE completion (Read After Write - RAW) which is unattractive, because it degrades user performance. To reduce the performance penalty associated with RAW, we propose to retain the written content in the disk cache and verify it once the disk drive becomes idle. Although attractive, this approach (called IRAW - Idle Read After Write) contends for resources, i.e., cache and idle time, with user traffic and other background activities. In this paper, we present a trace-driven evaluation of IRAW and show its feasibility. Our analysis indicates that idleness is present in disk drives and can be utilized for WRITE verification with minimal effect on user performance. IRAW benefits significantly if some amount of cache, i.e., 1 or 2 MB, is dedicated to retain the unverified WRITES. If the cache is shared with the user requests then a cache retention policy that places both READs and WRITES upon completion at the most recently used cache segment, yields best IRAW performance without effecting user READs cache hit ratio and overall user performance.

1 Introduction

Nowadays the majority of the available information is digitally stored and preserved. As a result, it becomes critically important that this vast amount of data is avail-

able and accurate anytime it is accessed. Storage systems that host digitally stored data strive to achieve data availability and consistency. Data availability is associated mostly with hardware failures and redundancy is the common approach to address it. Data consistency is associated with hardware, firmware, and software errors. Redundancy is not sufficient to protect the data from corruption and sophisticated techniques, including checksumming, versioning, and verification, need to be in place throughout the storage hierarchy to avoid, detect, and successfully correct errors that cause data inconsistencies [9].

Generally, faults that affect data availability [20, 14] occur more often than errors that cause data corruption [2]. Consequently, data availability [13, 10] has received wider attention on storage design than data consistency [9]. Recent evaluation of a large data set [2] shows that the probability of an enterprise-level disk experiencing data corruption is low, i.e., only 0.06%. Nevertheless, when considering the large amount of digitally stored data, the occurrence rate of data corruption becomes non-negligible. As a result, ensuring data consistency has gained wide interest among storage system designers [2, 9, 12].

Data corruption often occurs during the WRITE process somewhere in the IO path. Consequently, techniques that avoid, detect, and correct data corruption are commonly associated with the management of WRITES. Examples include the log-structured and journaling file systems [19, 22], data checksumming and identification at the file system level (i.e., ZFS) or controller level [12, 15], as well as WRITE verification anywhere in the IO path.

Traditionally, Read After Write (RAW) ensures the correctness of a WRITE by verifying the written content via an additional READ immediately after the WRITE completes. RAW degrades user performance significantly because it doubles the service time of WRITES. As a result,

RAW is activated at the disk drive level only during special circumstances, such as high temperatures, that may cause more WRITE errors. In this paper, we propose an effective technique to conduct WRITE verification at the disk drive level. Specifically, we propose Idle Read After Write - IRAW, which retains the content of a *completed* and *acknowledged* user WRITE request in the disk cache and verifies the on-disk content with the cached content during idle times. Using idle times for WRITE verification reduces significantly the negative impact this process has on the user performance. We show the effectiveness of IRAW via extensive trace-driven simulations.

Unlike RAW, IRAW requires resources, i.e., cache space and idle time to operate efficiently at the disk level. Cache space is used to retain the unverified WRITES until the idle time becomes available for their verification. Nevertheless, in-disk caches of 16MB and underutilized disks (as indicated by disk-level traces) enable the effective operation of a feature like IRAW.

IRAW benefits significantly if some amount (i.e., 2 MB) of dedicated cache is available for the retention of the unverified WRITES. Our analysis shows that even if the cache space is fully shared between the user traffic and the unverified WRITES, a cache retention policy that places both READs and WRITES at the most-recently-used position in the cache segment list, yields satisfactory IRAW performance, without affecting READ cache hit ratio, and consequently, user performance. We conclude that IRAW is a feature that with a priority similar to “best-effort” enhances data consistency at the disk drive level, because it validates more than 90% of all the written content even in the busiest environments.

The rest of the paper is organized as follows. Section 2 discusses the causes of data corruption and focuses on data corruption detection and correction at the disk drive level. We describe the WRITE verification process in Section 3. Section 4 describes the disk-level traces used in our evaluation and relates their characteristics to the effectiveness of detection and correction of data corruption at the disk drive level. In Section 5, we present a comprehensive analysis of WRITE verification in idle time and its effectiveness under various resources management policies. Section 6 presents a summary of the existing work on data availability and reliability, in general, and data consistency, in particular. We conclude the paper with Section 7, which summarizes our work.

2 Background

In this section, we provide some background on data corruption and ways to address it at various levels of the IO path. Generally, data corruption is caused during the

WRITE process because of various causes. Data corruption occurs when a WRITE, even if acknowledged as successful, is erroneous. WRITE errors may lead to data being stored incorrectly, partially, or not in the location where it is supposed to be [9]. These WRITE errors are known as lost WRITES, torn WRITES, and misdirected WRITES, respectively. The cause of such errors may be found anywhere in the storage hierarchy.

Traditionally, data inconsistencies have been linked with the non-atomicity of the file system WRITES [19, 22]. A file-system WRITE consists of several steps and if the system crashes or there is a power failure while these steps are being carried out, the data may be inconsistent upon restarting the system. Legacy file systems such as log-structured and journaling file systems address data inconsistencies caused by system crashes and power failures [19, 22].

However, data corruption may be caused during the WRITE process by errors (bugs) in the software or firmware throughout the IO path, from the file system to the disk drives, or by faulty hardware. Although erroneous, these WRITES are acknowledged as successful to the user. These errors are detected only when the data is accessed again and as a result these errors cause *silent* data corruption. WRITE errors that cause silent data corruption are the focus of this paper. Addressing data inconsistencies because of power failures or system crashes are outside the scope of our paper.

Errors that cause silent data corruption represent a concern in storage system design, because, if left undetected, they may lead to data loss or even worse deliver inaccurate data to the user. Various checksumming techniques are used to detect and correct silent data corruption in the higher levels of the IO hierarchy. For example, ZFS [12] uses checksumming to ensure data integrity and consistency. Similarly, at the storage controller level checksumming techniques are coupled with the available data redundancy to further improve data integrity [9]. Logical background media scans detect parity inconsistencies by accessing the data in a disk array and building and checking the parity for each stripe of data [1].

Disk drives are responsible for a portion of WRITE errors that may cause silent data corruption in a storage system. WRITE errors at the disk drive may be caused by faulty firmware or hardware. The written content is incorrect although the completion of the WRITE command is acknowledged as successful to the user. Disk drives can detect and correct the majority of the disk-level WRITE errors via WRITE verification. In particular, disk drives can detect and correct WRITE errors when data is written incorrectly, partially or not at all at a specific location. WRITE verification at the disk level

does not help with misdirected WRITES, where the content is written somewhere else on the disk or on another disk in a RAID array.

3 Disk-level WRITE Verification

At the disk level, WRITE errors can be detected and recovered by verifying that the WRITE command was really successful, i.e., by comparing the written content with the original content in the disk drive cache. If inconsistency is found, then the data is re-written. WRITE verification can be conducted only if the written data is still in the disk cache. As a result, WRITE verification can occur immediately upon completion of a WRITE or soon thereafter. If the verification occurs immediately upon a WRITE completion, the process is known as WRITE Verify or Read-After-Write (RAW). RAW has been available for a long time as an *optional* feature in the majority of hard drives. Its major drawback is that it requires one additional READ for each WRITE, doubling the completion time of WRITES (in average). Consequently, RAW is turned on only if the drive operates in extreme conditions (such as high temperature) when the probability of WRITE errors is high.

If the recently written data is retained in the disk cache even after a WRITE is completed, then the disk may be able to verify the written content at a more opportune time, such as the disk idle times (when no user requests are waiting for service). This technique is called Idle READ After WRITE (IRAW). Because disk arm seeking is a non-instantaneously preemptable process, the user requests will be delayed even if verifications happen in idle time, albeit the delay is much smaller than under RAW. As a result IRAW represents a more attractive option to WRITE verification at the disk drive level than RAW.

There is a significant difference between RAW and IRAW with regard to the resource requirements these two features have. RAW does not require additional resources to run, while IRAW is enabled only if there are resources, namely cache and idle time, available at the disk drive. The main enabler for IRAW in modern disk drives is the large amount of the available in-disk cache. The majority of disk drives today have 16 MB of cache space. The existence of such amount of cache enables the drive to retain the recently written data for longer, i.e., until the disk drive becomes idle, when the WRITE verification causes minimal performance degradation on user performance.

The effectiveness of IRAW depends on effective management of the available cache and idle time. Both cache and idle time represent resources that are used exten-

sively at the disk drive, and IRAW will contend with other features and processes to make use of them both. For example, in-disk cache is mainly used to improve READ performance by exploiting the spatial and temporal locality of the workload, i.e., aggressively prefetching data from the disk or retaining recent READs in the cache hoping that incoming requests will find the data in the cache and avoid costly disk accesses. On the other hand, idle time is often used to deploy features that enhance drive operation such as background media scans. IRAW should not fully utilize the idle time and limit the execution of other background features.

On average, disk drives exhibit low to moderate utilization [17], which indicates that idle intervals will be available for WRITE verifications. Furthermore, in low and moderate utilization, busy periods are short as well. As a result only a few WRITES will need to be retained in the cache, and wait for verification during the incoming idle period. Consequently, IRAW cache requirements are expected to be reasonable. However, the disk drive workloads are characterized by bursty periods [18] which cause temporal resource contention and inability to complete WRITE verifications. In this paper, we focus on the evaluation of IRAW and ways to manage resources, i.e., cache and idle time, such that IRAW runs effectively, i.e., the highest number of WRITES is verified with minimal impact on user performance. Our focus is on four key issues:

- the available idle time for IRAW,
- the impact of IRAW on the performance of user requests, because they arrive during a non-preemptive WRITE verification,
- the cache requirements that would enable IRAW to verify more than 90% of all WRITES in the workload,
- the impact that retention of unverified WRITES in the cache has on READ cache hit ratio.

4 Trace Characterization

The traces that we use to drive our analysis are measured in various enterprise systems. These systems run dedicated servers that are identified by the name of the trace. Specifically, we use five traces in our evaluation; the “Web” trace measured in a web server, the “E-mail” trace measured in an e-mail server, the “Code Dev.” trace measured in a code development server, the “User Acc.” trace measured in a server that manages the home directory with the accounts of the users in the system, and the

Trace	Length (hrs)	Idle %	Avg. Idle Int. (ms)	R/W Ratio
Web	7	96	274	44/56
E-mail	25	92	119	99/1
User Acc.	12	98	625	87/13
Code Dev.	12	94	183	88/12
SAS	24	99	88	40/60

Table 1: General characteristics for disk-level traces used in our analysis.

“SAS” trace measured in a server running the SAS statistical package. Several of the measured storage subsystems consist of multiple disks, but throughout this paper, we focus on traces corresponding to the activity of single disks. Traces record several hours of disk-level activity (see Table 1) which make them representative for the purpose of this evaluation.

Traces record for each request the disk arrival time (in ms), disk departure time (in ms), request length (in bytes), request location (LBA), and request type (READ or WRITE). Here, we focus mostly on characteristics that are relevant to IRAW. General characterization of the traces as well as how they were collected can be found in [17, 18]. The only information we have on the architecture of the measured systems is the dedicated service they provide and the number of disks hosted by the storage subsystem.

Several trace characteristics such as arrival rate, READ/WRITE ratio, idle and busy time distributions are directly related to the ability of the disk drive to verify WRITES during idle intervals. In Table 1, we give the general characteristics (i.e., trace length, disk idleness, average length of idle intervals, and READ/WRITE ratio) of the traces under evaluation. While READ/WRITE ratio is derived using only the information on the *request type* column of each trace, the idleness and idle interval lengths are calculated from the information available in the arrival time and departure time columns. The calculation of system idleness as well as the length of idle and busy periods from the traces is exact (not approximate), and facilitates accurate evaluation of IRAW.

Table 1 indicates that disk drives are mostly idle, which represents a good opportunity for IRAW to complete successfully during idle times. The average length of idle intervals indicates that several WRITES may be verified during each idle interval. The READ/WRITE ratio in the incoming user traffic indicates the portion of the workload that needs verification in idle times and determines the IRAW load. Because the READ/WRITE ratio varies in the traces of Table 1, the IRAW performance will be evaluated under different load levels. Although

the application is the main determining factor of the READ/WRITE ratio of disk-level workloads, the storage system architecture plays an important role as well. For the systems where the Web and the SAS traces were measured, the IO path has less resources and, consequently, intelligence than the other three traces. We came to this conclusion because the Web and SAS traces are measured on storage subsystems with single disks while the other traces are measured on storage subsystems with multiple disks. This leads us to believe that, except the Web and the SAS systems, the measured storage subsystems are organized in RAID arrays. Also from the traces, we can extract information on the WRITE optimization that takes place above the disk level. WRITE optimization consists of coalescing, usage of non-volatile caches, and other features, which reduce overall WRITE traffic. An indication, at the disk level, of the presence of non-volatile caches or other WRITE optimization features in the IO path, (see [17] for longer discussion), is the frequency of re-writes on a recently written location. While for the E-mail, User Acc. and Code Dev. traces the written locations are not re-written for the duration of each trace, for the Web and SAS traces this is not the case.

Figure 1 gives the arrival rate (i.e., the number of requests per second) as a function of time for several enterprise traces from Table 1. The disk-level workload is characterized by bursts in the arrival process. The arrival bursts are sometimes sustained for long (i.e., several minutes) periods of time. Arrival bursts represent periods of time when resources available for IRAW (i.e., cache and idle time) are limited. Consecutively, it is expected that IRAW will not have enough resources to verify all WRITES in an environment with bursty workloads.

In Figure 2, we present the distribution of idle periods for the traces of Table 1. In the plot, the x-axis is in log-scale to emphasize the body of the distribution that indicates the common length of the idle intervals. Almost 40% of the idle intervals in the traces are longer than 100 ms and only one in every three idle intervals is less than a couple of milliseconds. Such idle time characteristics favor IRAW and indicate that in each idle interval, the drive will be able to verify at least several WRITES.

The minimum length of the idle intervals, as well as their frequency is a useful indicator in deciding the *idle waiting* period, i.e., the period of time during which the drive remains idle although IRAW can be performed. Idle waiting is a common technique to avoid utilizing very short idle intervals with background features like IRAW and to minimize the effect disk-level background features have on user performance [4]. The case when a new user request arrives while a WRITE is being verified represents the case when IRAW degrades the performance of user requests. Figure 2 clearly indicates that

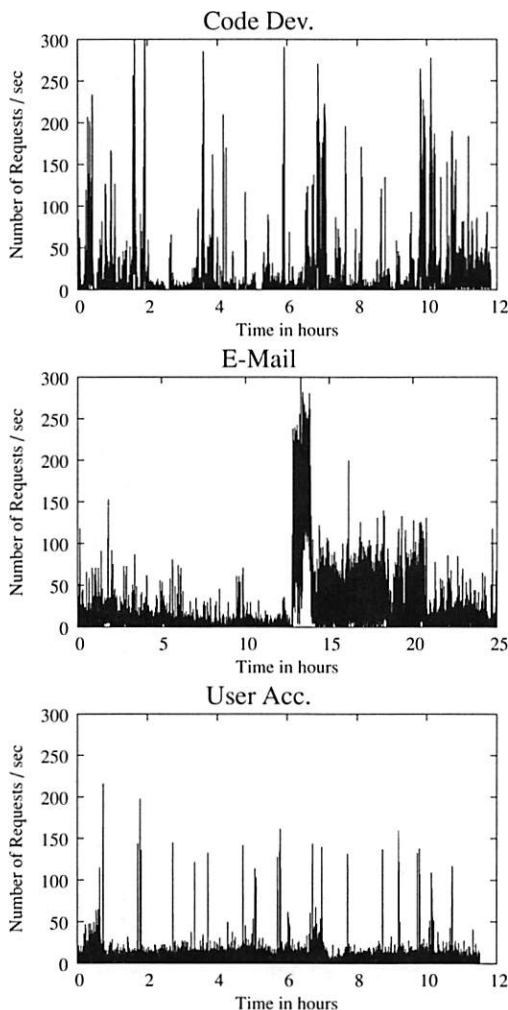


Figure 1: Arrival rate, measured in number of requests per second, as a function of time for several traces from Table 1. The arrivals are bursty in enterprise systems.

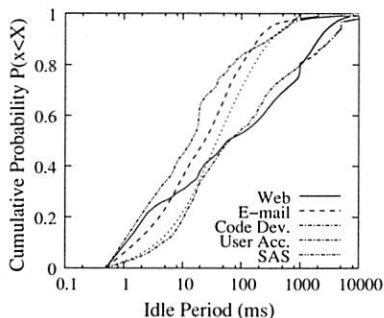


Figure 2: Distribution of idle periods for the traces of Table 1. X-axis is in log scale. The higher the line the shorter the idle periods are for the specific trace.

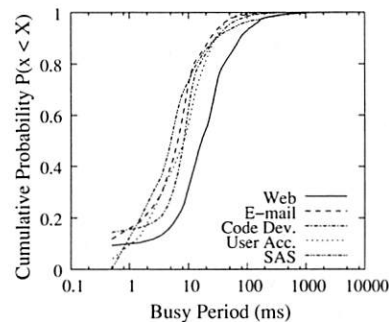


Figure 3: Distribution of busy periods for the traces of Table 1. The x-axis is in log-scale. The higher the line the shorter the busy periods are for the specific trace.

more than 90% of all idle intervals in all evaluated traces are longer than 10 ms, which leads us to optimistically state that by waiting a couple of milliseconds in an idle drive before a WRITE verification starts, the impact on the user requests performance will be contained to minimum.

IRAW effectiveness depends not only on the available idleness and length of idle periods, but also on the length of the busy periods at the disk drive level. The longer the busy period the larger the number of unverified WRITES waiting for the next idle period and occupying cache space. In Figure 3, we present the distributions of busy periods for the traces of Table 1. Similarly to Figure 2, the x-axis of the plots is in log-scale. The distribution of the length of busy periods indicates that disk busy times are relatively short. Across all traces only 1% of busy periods are larger than 100 ms. The shape of the busy period distribution suggests that most WRITES will get the chance to be verified during the idle period that immediately follows a busy period. Moreover, short busy intervals (Figure 3) and long idle intervals (Figure 2) indicate that IRAW will only use a fraction of the available idle time leaving room for additional background activities to be carried out too.

5 Evaluation of IRAW

The evaluation of IRAW is driven by the traces introduced in the previous section. Initially, we define a simplified version of IRAW, where (1) each WRITE verification takes the same amount of time, i.e., 5 ms, (2) there is dedicated cache available to store unverified WRITES, and (3) the length of the idle interval is known, which means that WRITE verification will not affect the incoming user requests. With these assumptions, we can evaluate the effectiveness of IRAW directly from the traces and obtain an approximate estimation of the resource re-

quirements for IRAW. We refer to this part of the evaluation as *trace-based* and discuss it in Section 5.1.

We further develop a simulation model for IRAW under the DiskSim 2.0 [6] disk-level simulator to relax the above assumptions and take into consideration the cache management aspect of IRAW. The simulation model is driven by the same set of traces. Because the simulation model represents an open model, we do not use the departure time field from the traces. As a result, the simulation model does not follow the idle and busy periods of the traces. The idle and busy periods in the simulation model are determined by the simulated disk, cache management policy, and the available cache size. We refer to this part of the evaluation as *simulation-based* and discuss it in Section 5.2.

In our evaluation, the efficiency of IRAW is measured by the *IRAW validation rate*, which represents the portion of WRITE requests verified during idle times. Any IRAW validation rate less than 100% indicates that not all WRITES are verified. A WRITE is left unverified if it is evicted from the cache before idle time becomes available to verify it. Limited cache and/or limited idle time cause the IRAW validation rate to be less than 100%.

5.1 Trace-based Analysis

In the trace-based analysis, we assume full knowledge of the idle time duration, which means that IRAW will have no impact on the user performance for this type of analysis. We assume the validation of each WRITE takes the same amount of time to complete, i.e., 5 ms - the average time to complete a request at the disk drive. An unverified WRITE corresponds to the same WRITE request originally received by the drive, i.e., no coalescing or other techniques are used to reduce the number of unverified WRITES. Verification is done in FCFS fashion.

Initially, we pose no restriction on the amount of available cache at the disk drive level. This assumption, although unrealistic, helps with the estimation of the maximum amount of cache required by IRAW to verify all WRITES in the user workload. However, we do limit the amount of time an unverified WRITE waits in the cache for verification. We refer to this threshold as $IRAW_{Age}$ and measure it in number of idle intervals. An unverified WRITE waits through at most $IRAW_{Age}$ idle intervals before it is evicted from the cache. The threshold $IRAW_{Age}$ measures, indirectly, idle time availability at the disk drive level. That is, if a WRITE remains unverified through $IRAW_{Age}$ idle intervals, then, most probably, it will remain unverified in a more realistic scenario with limited cache space. The larger the $IRAW_{Age}$, the larger the maximum cache space require-

Trace	IRAW Rate	$IRAW_{Age}$	Max Cache
Web	97 %	512	22 MB
E-mail	100 %	32	0.4 MB
User Acc.	100 %	64	1.7 MB
Code Dev.	100 %	256	8 MB
SAS	95 %	512	50 MB

Table 2: IRAW Verification Rate assuming unlimited cache and average verification time of 5 ms.

ments and the higher the IRAW validation rate.

We set $IRAW_{Age}$ threshold to be 512, which means that the disk will retain an unverified WRITE through no more than 512 idle intervals. We measure the IRAW verification rate as a function of the $IRAW_{Age}$ and estimate the maximum amount of cache required to retain the unverified WRITES until verification. We present our findings in Table 2.

Table 2 indicates that IRAW validation rate for 60% of the traces is 100%, with only moderate cache requirements, i.e., up to 8 MB of cache. For the traces that achieve 100% IRAW validation rate (i.e., E-mail, User Acc. and Code Dev.), the $IRAW_{Age}$ value is below the threshold of 512. This shows that, for these traces, there is idle time available to verify all WRITES in the workload. From Table 1, we notice that the three traces that achieve 100% validation rate with moderate cache requirements have the lowest number of WRITES in the workload. The other two traces, namely Web and SAS, have many more WRITES in their workload mix. As a result, the verification rate is not 100%. Nevertheless, the Web and SAS traces achieve at least 95% IRAW validation rate. For these two traces, the amount of required cache space is high, i.e., more than 20 MB, which is unrealistic for a disk drive today. Following the discussion in Section 4 about the READ/WRITE ratio of traces in Table 1, recall that the high READ/WRITE ratio for Web and SAS may be associated with the IO path hierarchy in the systems where these traces were collected.

The results in Table 2, give a high level indication that the IRAW may be an effective feature, which will restrict performance degradation for user requests while maintaining high level of WRITE verification. However, because IRAW requires both cache and idle time to complete the verifications, the ratio of verified WRITES, is not expected to be 100% in all cases.

The assumption of having unlimited cache is unrealistic. Hence, in the next experiment, we assume that the dedicated cache to IRAW is only 8 MB. By limiting the available cache size the $IRAW_{Age}$ threshold is eliminated, because now the reason for a WRITE to remain

Trace	Web	E-mail	User Acc.	Code Dev.	SAS
IRAW Rate	91%	100%	100%	100%	91%

Table 3: IRAW Verification Rate assuming 8 MB of available cache and average verification time of 5 ms.

unverified is the lack of cache to store it rather than the lack of idle time.

The corresponding results are presented in Table 3. As expected from the results in Table 2, IRAW verification rate for the E-mail, User Acc. and Code Dev. traces is still 100%. The other two traces, i.e., Web and SAS, perform slightly worse than in the case of unlimited cache (see Table 2). The Web and SAS traces require more than 20MB of cache space to achieve at least 95% IRAW verification rate. With only 8MB, i.e., almost three times less cache, IRAW validation rate is at least 91%. This result indicates that the maximum cache space requirement is related to bursty periods in the trace that reduce the availability of idle time for IRAW. Consequently, even in bursty environments where resources may be limited at time, there are opportunities to achieve high IRAW verification rates, i.e., above 90%.

5.2 Simulation-based Analysis

We use DiskSim 2.0 disk-level simulation environment [6] to evaluate in more detail the cache management strategies that work for IRAW. The simulation is driven by the same set of traces that are described in Section 4. The trace-based analysis provided an approximate estimation of IRAW cache space requirements, idleness requirements, as well as the overall IRAW validation rate. Section 5.1 concluded that in the enterprise environment, IRAW verifies at least 90% of WRITES with moderate resource requirements (i.e., 8MB of cache) dedicated to IRAW.

The following simulation-based analysis intends to evaluate in more detail the cache management policies and how they effect IRAW performance and user request performance in presence of IRAW. The simulated environment is more realistic than the trace-based one, where several assumptions were in place. For example, in the simulation-based analysis, the idle interval length is not known beforehand and the verification time for WRITES is not deterministic. Consequently, during the verification of a WRITE a user request may arrive and be delayed because the WRITE verification cannot be preempted instantaneously.

We simulate two disks, one with 15K RPM and 73GB of space and the second one with 10K RPM and 146GB of

Trace	Max Cache	IRAW Rate	IRAW Response Time
Web	60 MB	100%	283 ms
E-mail	0.7 MB	100%	8 ms
User Acc.	2 MB	100%	10 ms
Code Dev.	60 MB	100%	5435 ms
SAS	48 MB	100%	1120 ms

Table 4: IRAW maximum cache requirements, verification rate, and verification response time, in our simulation model with unlimited cache space for unverified WRITES.

space, which model accurately the disks where the traces were measured. The latter disk is used to simulate only the Code Dev. trace from Table 1. Both disks are set to have an average seek time of 5.4 ms. The requests in both foreground and background queue are scheduled using the Shortest Positioning Time First (SPTF) algorithm. The IRAW simulation model is based on the existing components of the disk simulation model in DiskSim 2.0. The queue module in DiskSim 2.0 is used to manage and schedule the unverified WRITES, and the cache module is used to manage the available cache segments between the user READs and WRITES and the unverified WRITES.

As previously discussed, the trace-driven simulation results would reflect the modeling of scheduling, caching, and serving of user requests and will not fully comply with the results obtained from a trace-based evaluation only approach. Consequently, we do not expect exact agreement between the results in the trace-based evaluation of Subsection 5.1 and the simulation-based evaluation in this subsection.

Once the disk becomes idle, the WRITE verification process starts after 1 ms of idle time has elapsed. WRITE verifications are scheduled after some idle time has elapsed at the disk level to avoid utilizing the very short idle intervals and, consequently, limit the negative effect WRITE verification may have on user request performance. The benefit of idle waiting in scheduling low-priority requests such as WRITE verifications under IRAW are discussed in [4, 11].

Initially, we estimate the maximum cache requirement for each of the traces under the simulation model. For this the simulation is run with no limitation on cache availability. The goal is to estimate how much cache is necessary to achieve 100% WRITE verification rate. Recall that the longer the unverified WRITES are allowed to wait for validation the larger the required cache space to store them. The simulation results are presented in Table 4.

The results in Table 4 show that only for two traces (40% of all evaluated traces), IRAW achieves 100% validation rate by requiring a maximum of 2MB cache space. These two traces are characterized by low disk utilization (i.e., 99% idleness) or READ dominated workload (i.e., the E-mail trace has only 1.3% WRITES). The other subset of traces (60% of them) requires more than 48MB of cache space, in the worst case, to achieve 100% IRAW verification rate. The worst WRITE verification response time in these traces is 5.4 sec, which explains the large cache requirements. The results of Table 4 are qualitatively the same as the one Table 2. IRAW verification rate of 100% comes with impractical cache requirements for half of the traces.

In an enterprise environment, IRAW is expected to require large cache space in order to achieve 100% IRAW validation rate, because the workload, as indicated in Section 4, is characterized by bursts. The bursts accumulate significant amount of unverified WRITES in short periods of time. These WRITES need to be stored until the burst passes and the idleness facilitates the verification.

Table 4 shows also the average IRAW response time, i.e., the time unverified WRITES are retained in the cache. For the traces that capture light load, i.e., E-mail and User Acc. traces, the WRITES are verified without waiting too long, similar to how RAW would perform. For the traces that capture medium to high load, i.e., Code Dev. and SAS traces, the IRAW response time is up to several seconds, which indicates that the unverified WRITES will occupy the available cache for relatively long periods of time.

Although IRAW is designed to run in background, it will, unavoidably, impact at some level the performance of the user requests, i.e., foreground work. There are two ways that IRAW degrades foreground performance

- Upon arrival, a new request finds the disk busy verifying a WRITE when otherwise the disk would have been idle. Because the WRITE validation cannot be interrupted once started, the response time of the newly arrived user request and of any other user requests in the incoming foreground busy period will be longer by the amount of time between the first user requests arrival and the completion of WRITE verification. The WRITE verification as any other disk-level service is non-instantaneously preemptable because seeking in the disk drive is non-preemptable.
- Unverified WRITES are stored in the disk cache to wait for an idle period when they can be verified. As a result, the unverified WRITES occupy cache

Trace	Idleness	R/W Ratio	Max. diff	Avg. IOPS diff.
Web	96 %	44/56%	0.53%	0.02%
E-mail	92 %	99/1 %	0.11%	0.00%
User Acc.	98 %	87/13%	0.02%	0.00%
Code Dev.	94 %	88/12%	2.37%	0.08%
SAS	99 %	40/60%	0.12%	0.00%

Table 5: IRAW impact on system throughput measured by IOPS.

space, which otherwise would have been used by the user READ requests. As a consequence, IRAW may reduce READ performance by reducing the READ cache hit ratio.

We analyze the impact of IRAW on the user performance by quantifying the reduction in the user throughput (measured by IOs per second - IOPS) and the additional wait experienced by the user requests because of the non-preemptability of WRITE verifications. We present our findings regarding the system throughput in Table 5 and the IRAW-caused delays in the user requests response time in Figure 4.

The trace-driven simulation model represents an open system. As a result the arrival times are fixed and will not change if the model simulates a disk slowed down by the presence of IRAW. This means that independent of the response time of requests, all requests will be served by the disk drive more or less within the same time period overall. This holds, particularly, because the traces represent cases with low and moderate utilization. As a result, to estimate the impact IRAW has on IOPS, we estimate the metric over short periods of time rather over the entire trace (long period of time) and focus on differences between the IOPS when IRAW is present and when IRAW is not present at the disk-level. We follow two approaches to estimate the IRAW caused degradation in IOPS. First we calculate the IOPS over 5 min intervals and report the worst case, i.e., the maximum IRAW-caused degradation in the IOPS over a 5 minutes interval. Second we calculate the IOPS for each second and report the average on the observed degradation. In both estimation methods, the impact that IRAW has on IOPS is low. We conclude that IRAW has minimal effect on system throughput for the evaluated traces from Table 5.

Results of Table 5 are confirmed by the distribution of the IRAW caused delays in the response time of user requests. The majority of user requests are not delayed by IRAW, as clearly indicated in Figure 4. For all traces, only less than 10% of user requests are delayed a few milliseconds, because they find the disk busy verifying

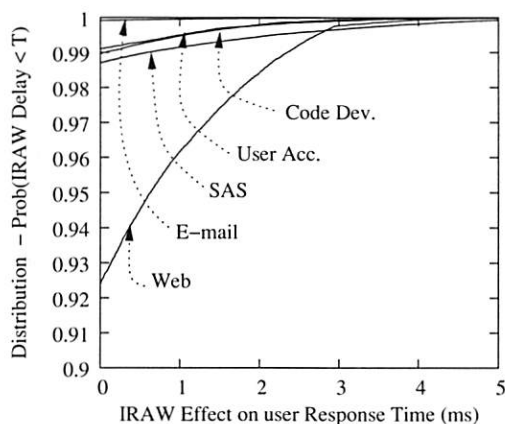


Figure 4: Distribution of IRAW caused delays.

WRITES. For some traces such as the E-mail one, the delays are virtually non-existent. Since the average verification time is only a few milliseconds, the maximum IRAW-caused delays are also a couple of milliseconds as indicated by the x-axis of Figure 4.

In order to minimize the impact IRAW has on user performance, it is critical for IRAW to start WRITE verification only after some idle time has elapsed, called *idle wait*. In Figure 5, we show the IRAW validation rate for three different traces, as a function of cache size and length of the idle wait. The results suggest that an idle wait of up to 5 ms does not reduce the IRAW verification rate and does not affect the user requests performance. In our simulation model, we use the idle IRAW wait of 1 ms, but anything close to the average WRITE verification time of 3 ms yields similar performance.

5.3 Cache management policies

Disk drives today have approximately 16 MBytes of cache available. Disk caches are used to reduce the disk traffic by serving some of requests from the cache. The disk cache is volatile memory and because of data reliability concerns it is used to improve READ rather than WRITE performance by aggressive prefetching and data retention.

As a result, for background features like IRAW, which require some amount of cache for their operation, efficient management of the available cache is critical. While in the previous sections, we focused on evaluating IRAW and its maximum cache requirements, in this subsection, we evaluate IRAW performance under various cache management policies. We also estimate the impact that IRAW has on the READ cache hit ratio, which is directly related to the user performance.

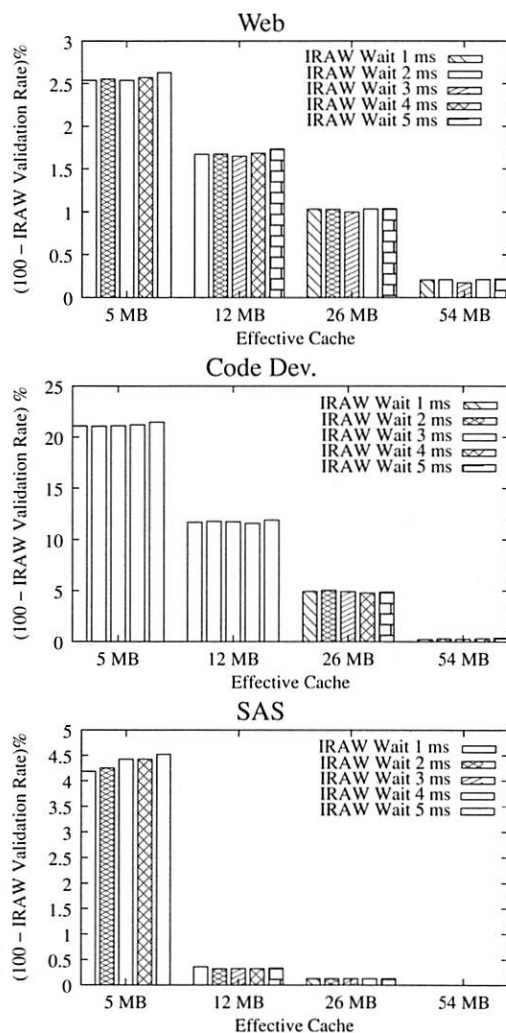


Figure 5: Impact of idle wait and cache space on IRAW performance.

There are two ways that IRAW uses the available cache space. First, IRAW shares the cache with the user READ traffic. In this case, READs and unverified WRITES contend for the cache, with READs having more or at least the same priority as the unverified WRITES. Second, IRAW uses dedicated cache space to store the unverified WRITES. The IRAW dedicated cache space should enhance IRAW performance by minimally affecting READ cache hit ratio.

If IRAW and the READ user traffic share the cache, by default, IRAW has a “best-effort” priority, i.e., the lowest possible priority, because this is the priority of completed user WRITES in the disk cache. This priority scheme gives no guarantees on IRAW verification rate. If some amount of dedicated cache space is allocated only for unverified WRITES, then the IRAW priority is higher than just “best-effort”. Under this scheme, user READ re-

quests will have less cache space available and, consequently, READ cache hit ratio will be lower. Overall, the IRAW validation rate is expected to be higher when dedicated cache space is allocated for unverified WRITES than when IRAW contends for the available cache space with the READ user traffic.

The cache space in a disk drive is organized as a list of segments (schematically depicted in Figure 6). The head of the list of segments is the position from where the data is evicted from the cache. The head position is called the *Least Recently Used - LRU* segment and it has the lowest priority among all the cache segments. The further down a segment is from the LRU position, the higher its priority is and the further in the future its eviction time is. The tail of the segment list represents the segment with the highest priority and the furthest in the future its eviction time. The tail position is referred to as the *Most Recently Used - MRU* position.

Commonly in disk drives, a READ is placed at the MRU position once the data is read from the disk to the cache, and a recently completed WRITE is placed at the LRU position. This policy indicates that for caching purposes, READs have the highest priority and WRITES have the lowest priority. This is because a recently completed WRITE is not highly probable to be read in the near future. When a new READ occupies the MRU position, the previous holder of the MRU position is pushed up one position reducing its priority and the time it will be retained in the cache. All other segment holders are pushed up with one position as well, resulting in the eviction of the data from the LRU position. If there is a cache hit and a new READ request is accessing data found in the cache, the segment holding the data is placed in the MRU position and there is no eviction from the cache.

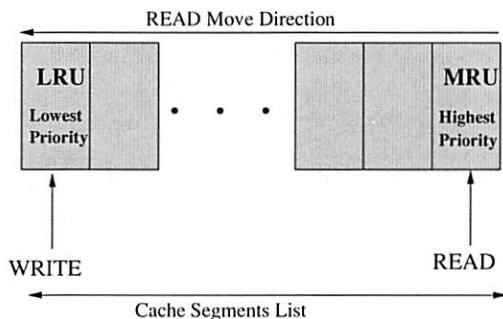


Figure 6: The model of the disk cache organized as a list of cache segments (each represented by a rectangle). The LRU position is the segment with the lowest retention priority and the MRU position is the segment with the highest retention priority. Upon completion, WRITES are placed at the LRU position and READS are placed at the MRU position.

The default cache retention policy does not favor the retention of unverified WRITES. As a result, in the following, we investigate how the available cache may be shared between the READ user traffic and the unverified WRITES such that both set of requests benefit from the available cache.

Initially, we evaluate the IRAW performance when it shares the cache space with the user READ traffic. We evaluate variations of the above default cache retention policy. A variation from the default cache retention policy is obtained by changing the default positions in cache for READs and unverified WRITES upon the user request completion. The following cache retention schemes are evaluated:

- *the default*; READs are placed in the MRU position and unverified WRITES in the LRU position (abbreviation: MRU/LRU),
- READs and unverified WRITES are both placed in the MRU position in a first-come-first-serve basis (abbreviation: MRU/MRU),
- READs and WRITES are left in their current segments upon completion, i.e., a WRITE is not moved to the LRU position, a READ cache hit is not moved to the MRU position, a READ miss is placed in the MRU position (abbreviation: -/-).

Note that any cache retention algorithm other than those which place WRITES in the LRU position upon completion, retain WRITES longer in the cache and occupy space otherwise used by READs, which consecutively reduces the READ cache hit ratio, even though minimally. This is the reason why in our evaluation, the READ cache hit ratio and the IRAW validation rate are the metrics of interest. We analyze them as a function of the available data cache size.

In Figure 7, we present the cache hit ratio as a function of the cache size for several traces and cache retention policies. The plots of Figure 7 suggest that it is imperative for the READ cache hit ratio to place READs in the MRU position once the data is brought from the disk to the cache (observe the poor cache hit ratio for the “-/-” cache retention policy which does not change the position of a READ upon a cache hit). The fact that WRITES are treated with higher priority by placing them into the MRU position too, leaves the READ cache hit ratio virtually unaffected. Another critical observation is that beyond some amount of available cache space, i.e., in all experiments approximately 12MB, the READ cache hit ratio does not increase indicating that adding extra cache space in a disk drive does not improve the READ cache hit ratio significantly, but can be used effectively for background features such as IRAW.

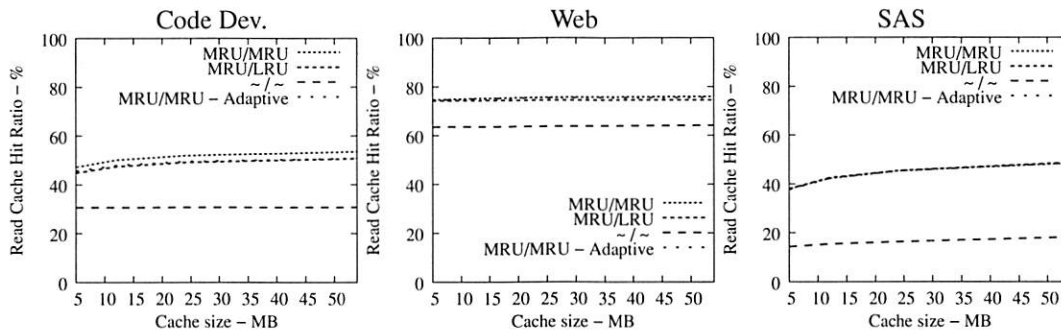


Figure 7: READ cache hit ratio as a function of cache size. Results are shown for various cache retention policies. A cache retention policy is identified by the placement of a READ (MRU, or no change) and the placement of unverified WRITES (MRU, LRU, or no change).

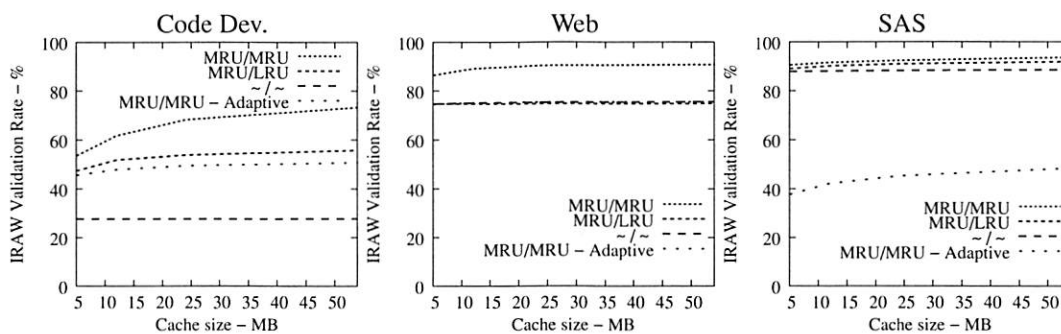


Figure 8: WRITE verification rate as a function of the cache size. Results are shown for various cache retention policies. A cache retention policy is identified by the placement of a READ (MRU, or no change) and the placement of unverified WRITES (MRU, LRU, or no change).

In Figure 8, we present the IRAW validation rate as a function of the available cache, under various cache retention policies for several traces. IRAW is more sensitive to the cache retention policy than the READ cache hit ratio (see Figure 7). Placing unverified WRITES in the MRU position is critical for the IRAW performance, in particular for the bursty case of the Code Dev. trace (recall that the simulated disk for the Code Dev. trace is a slower disk than for the rest of the enterprise traces). Figure 8 indicates that for most cases, i.e., 85% of them, shared cache retention algorithms work just fine and IRAW validation rate is above 90%.

In Figures 7 and 8, we also present results for an adaptive cache retention algorithm, where the READ/WRITE ratio of the workload is reflected on the amount of cache space used by READs and WRITES. For example, a READ/WRITE ratio of 70%/30% would cause 70% of the cache space to be used by READs and 30% by the unverified WRITES. As the ratio changes so does the usage of the cache. The adaptive policy improves the IRAW validation rate for most traces with almost no impact on READ cache hit ratio. However the gains are not substantial enough to justify the added complexity in

the implementation of the adaptive cache retention algorithm.

Figure 7 suggests that READ cache hit ratio does not increase significantly as the available cache size increases beyond a certain point, i.e., in our analysis it is 10-12 MB. Consequently, we evaluate the effectiveness of IRAW when some amount of dedicated cache is allocated for the retention of the unverified WRITES. In our evaluation, the user requests have the same amount of available cache for their use as well. For example, if IRAW will use 8MB of dedicated cache then so will the user READ requests. We present our results in Figure 9. Note that the plots in Figure 9 are the same as the respective ones in Figure 7 and Figure 8, but the “MRU/MRU - Adaptive” line is substituted with the “MRU/MRU - Dedicated” line. The results in Figure 7 indicate that the dedicated cache substantially improves the IRAW validation rate. This holds in particular for the heavy load cases such as the Code Dev. trace.

In conclusion, we emphasize that in order to maintain high READ cache hit ratio and high IRAW validation rate, it is critical for the available cache to be managed efficiently. Both READs and WRITES need to be placed

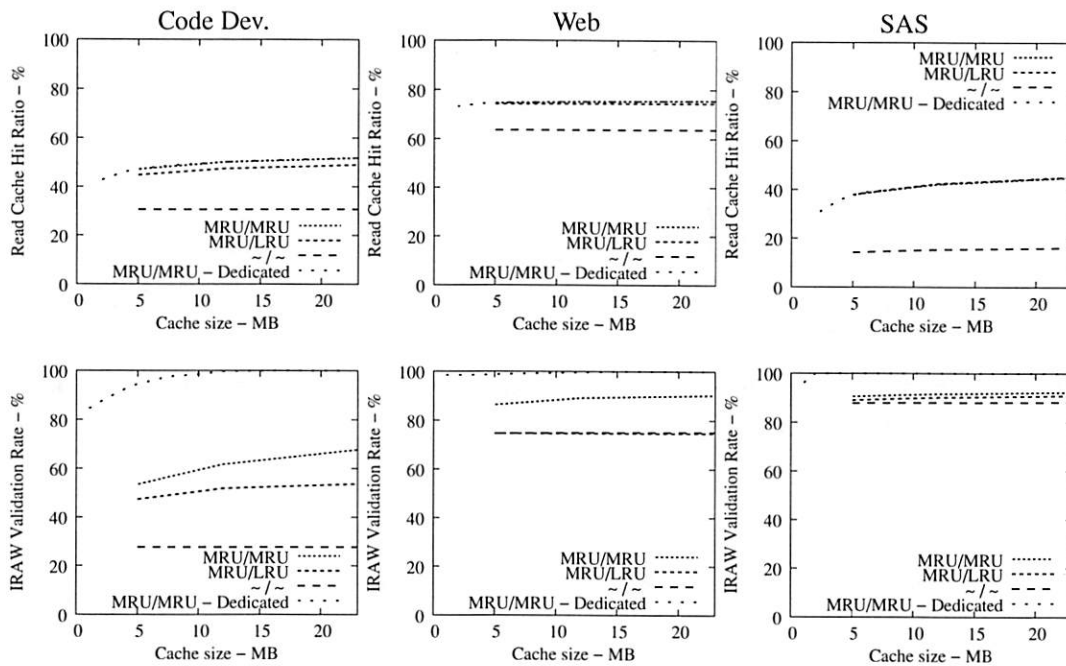


Figure 9: Read Cache hit ratio (first row) and IRAW validation rate (second row) as a function of the available dedicated cache.

in the MRU position upon completion. This cache retention policy yields the best performing IRAW for most environments, but not for the critical (very bursty) ones. The latter cases benefit enormously even if only a few MB of cache (i.e., 2 MB) are dedicated to store unverified WRITES. Additional dedicated cache space for IRAW (i.e., 4-12MB) yields the best IRAW validation rate in the evaluated environments.

6 Related Work

Although disk drive quality improves from one generation to the next, they represent complex devices that are susceptible to a variety of failures [23, 24]. Because drive failures may lead to data loss, storage systems have widely accepted the RAID architecture [13, 10], which protects the data from one or two simultaneous failures. In theory storage systems can be designed to protect from n simultaneous disk drive failures, if $m > n$ disks are available [16]. Contemporary storage systems have adopted a distributed architecture with multiple copies of any piece of data [7] for added reliability, while using inexpensive disk drives.

As the amount of digitally stored data increases, so does the significance of storage and drive failures [20, 14]. In particular, rare failure events have become more prevalent. For example, in recent years significant effort has been devoted to better understand the effect of latent

sector errors on overall data availability in storage systems [3, 5, 1]. Latent sector errors may happen at any time in a disk drive, but they may cause data loss (even of only a few sectors) if they remain undetected until another failure in the system (now with reduced data redundancy) triggers the entire data set to be accessed for reconstruction. To address such undesirable events, features like background media scans are added in storage system and disk drives [21, 1].

Traditionally, it has been the file system's task to ensure data consistency and integrity, assuming that the causes were related to power failure or system crashes during non-atomic WRITE operations. Legacy file systems address data consistency by implementing features like journaling and soft updates [22, 19]. Contemporary file systems [12, 15, 8] deploy more complex and aggressive features that involve forms of checksumming, versioning, identification for any piece of data stored in the system.

Today, storage system designers are concerned by silent data corruption. The growing complexity of systems enabling software, firmware, and hardware may cause data corruption and affect overall data integrity. Similar to disk latent sector errors, data corruption may happen at any time, but it can be detected only later on when the data is accessed. Such events may cause data loss, or, even worse, may deliver incorrect data to the user. Silent data corruption may occur in any component of the IO

path.

Recent results from a large field population of storage systems [2] indicate that the probability that a disk develops silent data corruption is low, i.e., only 0.06% for enterprise-level disks and 0.8% for near-line disks. This occurrence rate is one order of magnitude less than the rate of a disk developing latent sector errors. Detection of silent data corruption as well as the identification of its source is not trivial and various aggressive features are put in place throughout the IO path to protect against silent data corruption [9].

Silent data corruption is associated with WRITES and occurs when a WRITE, although acknowledged as successful, is not written in the media at all (i.e., lost WRITE), is written only partially (i.e., torn WRITE), or written in another location (i.e. misdirected WRITES). The disk drive may cause some of the above WRITE errors. Read-After-Write (RAW) detects and corrects some WRITE errors by verifying the written content with the cached content. RAW may be deployed at the disk drive level or array controller level. RAW degrades significantly user performance and this paper focuses on effective ways to conduct WRITE verification.

7 Conclusions

In this paper, we proposed Idle Read After Write (IRAW), which verifies WRITES at the disk drive level during idle time. IRAW aims at detecting and correcting any inconsistencies during the WRITE process that may cause silent data corruption and eventually data loss. Traditionally WRITE verification is conducted immediately after a WRITE completes via a process known as Read After Write. RAW verifies the content on the disk with the WRITE request in the disk cache. Because a WRITE is followed by an additional READ, RAW significantly degrades user's performance. IRAW addresses RAW's drawbacks by conducting the additional READs associated with a WRITE verification during idle time and minimizing the effect that WRITE verification has on user performance.

Unlike RAW, IRAW requires resources (i.e., cache and idle time) for its operation. Cache is required to store unverified WRITES until idle time becomes available to perform the WRITE verifications. Nevertheless, in-disk caches of 16MB and underutilized disks (as indicated by disk-level traces) enable the effective operation of a feature like IRAW. Although IRAW utilizes only idle times, it effects user request performance, because it contends for cache with the user traffic and it delays user requests if they arrive during the non-preemptable WRITE verification. Consequently, we measure the IRAW perfor-

mance by the ratio of verified WRITES and the effect it has on user request performance.

We used several disk-level traces to evaluate IRAW's feasibility. The traces confirm the availability of idleness at the disk-level and indicate that disk's operation is characterized by short busy periods and long idle periods, which favor IRAW. Via trace-driven simulations, we concluded that IRAW has minimal impact on the disk throughput. The maximal impact on disk throughput measured over 5 minutes intervals is less than 1% for the majority of the traces. The worst estimated disk throughput degradation among the evaluated traces is only 2%.

Our evaluation showed that the cache hit ratio for the user traffic (and consequently user performance) is maintained if both READs and WRITES are placed at the MRU (Most Recently Used) position in the cache upon completion. Because the READ cache hit ratio plateaus as the cache size increases, it is possible to use some dedicated cache space for IRAW without effecting READ cache hit ratio and improving considerably IRAW verification rate. Dedicated cache of 2MB seems to be sufficient to achieve as high as 100% IRAW validation rate for the majority of the evaluated traces. We conclude that IRAW is a feature that with a priority similar to "best-effort" enhances data integrity at the disk drive level, because it validates more than 90% of all the written content even in the burstiest environments.

References

- [1] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proceeding of the ACM SIGMETRICS*, pages 289–300, 2007.
- [2] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *to appear in Proceeding of the USENIX Annual Conference in File and Storage Systems*, 2008.
- [3] M. Baker, M. Shah, D. S. H. Rosenthal, M. Rousopoulos, P. Maniatis, T. J. Giuli, and P. P. Bungle. A fresh look at the reliability of long-term digital storage. In *EuroSys*, pages 221–234, 2006.
- [4] L. Eggert and J. D. Touch. Idletime scheduling with preemption intervals. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 249–262, Brighton, UK, Oct. 2005. ACM Press.

- [5] J. G. Elerath and M. Pecht. Enhanced reliability modeling of raid storage systems. In *DSN*, pages 175–184, 2007.
- [6] G. R. Ganger, B. L. Worthington, and Y. N. Patt. The DiskSim simulation environment, Version 2.0, Reference manual. Technical report, Electrical and Computer Engineering Department, Carnegie Mellon University, 1999.
- [7] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.
- [8] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherd-ing. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 283–296, Stevenson, Washington, October 2007.
- [9] A. Krioukov, L. N. Bairavasundaram, G. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity lost and parity regained. In *FAST*, 2008.
- [10] C. Lueth. RAID-DP: Network appliance implementation of RAID double parity for data protection. Technical report, Technical Report No. 3298, Network Appliance Inc, 2004.
- [11] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel. Efficient utilization of idle times. In *Proceedings of the ACM SIGMETRICS*, pages 371–372, 2007.
- [12] S. Mirosystems. Zfs: the last word in file systems. Technical report, <http://www.sun.com/2004-0914/feature>, 2004.
- [13] D. A. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference*, pages 109–116. ACM Press, 1988.
- [14] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, pages 17–28, 2007.
- [15] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.
- [16] M. . Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of ACM*, 36(2):335–348, 1989.
- [17] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference*, pages 97–103, May 2006.
- [18] A. Riska and E. Riedel. Long-range dependence at the disk drive level. In *Proceedings of the International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 41–50, 2006.
- [19] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transaction on Computer Systems*, 10(1):26–52, 1992.
- [20] B. Schroeder and G. A. Gibson. Understanding disk failure rates: What does an mttf of 1,000,000 hours mean to you? *ACM Transactions on Storage*, 3(3), 2007.
- [21] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the International Symposium on Modeling and Simulation of Computer and Communications Systems (MASCOTS)*. IEEE Press, 2004.
- [22] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. .A.Soules, and C. . Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceeding of the 2000 USENIX Annual Technical Conference*, 2000.
- [23] S. Shah and J. G. Elerath. Reliability analysis of disk drive failure mechanism. In *Proceedings of 2005 Annual Reliability and Maintainability Symposium*, pages 226–231. IEEE, January 2005.
- [24] J. Yang and F. Sun. A comprehensive review of hard-disk drive reliability. In *Proceeding of the IEEE Annual Reliability and Maintainability Symposium*, pages 403–409, 1999.

Design Tradeoffs for SSD Performance

Nitin Agrawal*, Vijayan Prabhakaran, Ted Wobber,
John D. Davis, Mark Manasse, Rina Panigrahy
Microsoft Research, Silicon Valley
**University of Wisconsin-Madison*

Abstract

Solid-state disks (SSDs) have the potential to revolutionize the storage system landscape. However, there is little published work about their internal organization or the design choices that SSD manufacturers face in pursuit of optimal performance. This paper presents a taxonomy of such design choices and analyzes the likely performance of various configurations using a trace-driven simulator and workload traces extracted from real systems. We find that SSD performance and lifetime is highly workload-sensitive, and that complex systems problems that normally appear higher in the storage stack, or even in distributed systems, are relevant to device firmware.

1 Introduction

The advent of the NAND-flash based solid-state storage device (SSD) is certain to represent a sea change in the architecture of computer storage subsystems. These devices are capable of producing not only exceptional bandwidth, but also random I/O performance that is orders of magnitude better than that of rotating disks. Moreover, SSDs offer both a significant savings in power budget and an absence of moving parts, improving system reliability.

Although solid-state disks cost significantly more per unit capacity than their rotating counterparts, there are numerous applications where they can be applied to great benefit. For example, in transaction-processing systems, disk capacity is often wasted in order to improve operation throughput. In such configurations, many small (cost inefficient) rotating disks are deployed to increase I/O parallelism. Large SSDs, suitably optimized for random read and write performance, could effectively replace whole farms of slow, rotating disks. At this writing, small SSDs are starting to appear in laptop computers because of their reduced power-profile and reliability in portable environments. As the cost of flash continues to decline, the potential application space for solid-state disks will certainly continue to grow.

Despite the promise that SSDs hold, there is little in the literature about the architectural tradeoffs inherent in

their design. Where such knowledge exists, it typically remains the intellectual property of SSD manufacturers. As a consequence, it is difficult to understand the architecture of a given device, and harder still to interpret its performance characteristics.

In this paper, we lay out a range of design tradeoffs that are relevant to NAND-flash solid-state storage. We then analyze several of these tradeoffs using a trace-based disk simulator that we have customized to characterize different SSD organizations. Since we can only speculate about the detailed internals of existing SSDs, we base our simulator on the specified properties of NAND-flash chips. Our analysis is driven by various traces captured from running systems such as a full-scale TPC-C benchmark, an Exchange server workload, and various standard file system benchmarks.

We find that many of the issues that arise in SSD design appear to mimic problems that have previously appeared higher in the storage stack. In solving these hard problems, there is considerable latitude for design choice. We show that the following systems issues are relevant to SSD performance:

- **Data placement.** Careful placement of data across the chips of an SSD is critical not only to provide load balancing, but to effect wear-leveling.
- **Parallelism.** The bandwidth and operation rate of any given flash chip is not sufficient to achieve optimal performance. Hence, memory components must be coordinated so as to operate in parallel.
- **Write ordering.** The properties of NAND flash present hard problems to the SSD designer. Small, randomly-ordered writes are especially tricky.
- **Workload management.** Performance is highly workload-dependent. For example, design decisions that produce good performance under sequential workloads may not benefit workloads that are not sequential, and vice versa.

As SSDs increase in complexity, existing disk models will become insufficient for predicting performance. In

particular, random write performance and disk lifetime will vary significantly due to the locality of disk write operations. We introduce a new model for characterizing this behavior based on cleaning efficiency and suggest a new wear-leveling algorithm for extending SSD lifetime.

The remainder of this paper is organized as follows. In the next section, we provide background on the properties of NAND-flash memory. Section 3 describes the basic functionality that SSD designers must provide and the major challenges in implementing these devices. Section 4 describes our simulation environment and presents an evaluation of the various design choices. Section 5 provides a discussion of SSD wear-leveling and gives preliminary simulator results on this topic. Related work is discussed in Section 6, and Section 7 concludes.

2 Background

Our discussion of flash memory is based on the latest product specifications for Samsung's K9XXG08UXM series NAND-flash part [29]. Other vendors such as Micron and Hynix offer products with similar features. For the remainder of this paper, we treat the 4GB Samsung part as a canonical exemplar, although the specifics of other vendors' parts will differ in some respects. We present the specifications for single-level cell (SLC) flash. Multi-level cell (MLC) flash is cheaper than SLC, but has inferior performance and lifetime.

Figure 1 shows a schematic for a flash package. A flash package is composed from one or more dies (also called chips). We describe a 4GB flash-package consisting of two 2GB dies, sharing an 8-bit serial I/O bus and a number of common control signals. The two dies have separate chip enable and ready/busy signals. Thus, one of the dies can accept commands and data while the other is carrying out another operation. The package also supports interleaved operations between the two dies.

Each die within a package contains 8192 *blocks*, organized among 4 *planes* of 2048 blocks. The dies can operate independently, each performing operations involving one or two planes. Two-plane commands can be executed on either plane-pairs 0 & 1 or 2 & 3, but not across other combinations. Each block in turn consists of 64 4KB *pages*. In addition to data, each page includes a 128 byte region to store metadata (identification and error-detection information). Table 1 presents the operational attributes of the Samsung 4GB flash memory.

2.1 Properties of Flash Memory

Data reads are at the granularity of flash pages, and a typical read operation takes 25 μ s to read a page from the media into a 4KB data register, and then subsequently shift it out over the data bus. The serial line transfers

Page Read to Register	25 μ s
Page Program (Write) from Register	200 μ s
Block Erase	1.5ms
Serial Access to Register (Data bus)	100 μ s
Die Size	2 GB
Block Size	256 KB
Page Size	4 KB
Data Register	4 KB
Planes per die	4
Dies per package (2GB/4GB/8GB)	1, 2 or 4
Program/Erase Cycles	100 K

Table 1: Operational flash parameters

data at 25ns per byte, or roughly 100 μ s per page. Flash media blocks must be *erased* before they can be reused for new data. An erase operation takes 1.5ms, which is considerably more expensive than a read or write operation. In addition, each block can be erased only a finite number of times before becoming unusable. This limit, 100K erase cycles for current generation flash, places a premium on careful block reuse.

Writing (or programming) is also done at page granularity by shifting data into the data register (100 μ s) and then writing it out to the flash cell (200 μ s). Pages must be written out sequentially within a block, from low to high addresses. The part also provides a specialized copy-back program operation from one page to another, improving performance by avoiding the need to transport data through the serial line to an external buffer.

In this paper, we discuss a 2 x 2GB flash package, but extensions to larger dies and/or packages with more dies are straightforward.

2.2 Bandwidth and Interleaving

The serial interface over which flash packages receive commands and transmit data is a primary bottleneck for SSD performance. The Samsung part takes roughly 100 μ s to transfer a 4KB page from the on-chip register to an off-chip controller. This dwarfs the 25 μ s required to move data into the register from the NAND cells. When these two operations are taken in series, a flash package can only produce 8000 page reads per second (32 MB/sec). If interleaving is employed within a die, the maximum read bandwidth from a single part improves to 10000 reads per second (40 MB/sec). Writes, on the other hand, require the same 100 μ s serial transfer time per page as reads, but 200 μ s programming time. Without interleaving, this gives a maximum, single-part write rate of 3330 pages per second (13 MB/sec). Interleaving the serial transfer time and the program operation doubles the overall bandwidth. In theory, because there are two independent dies on the packages we are considering, we can interleave three operations on the two dies put together. This would allow both writes and reads to progress at the speed of the serial interconnect.

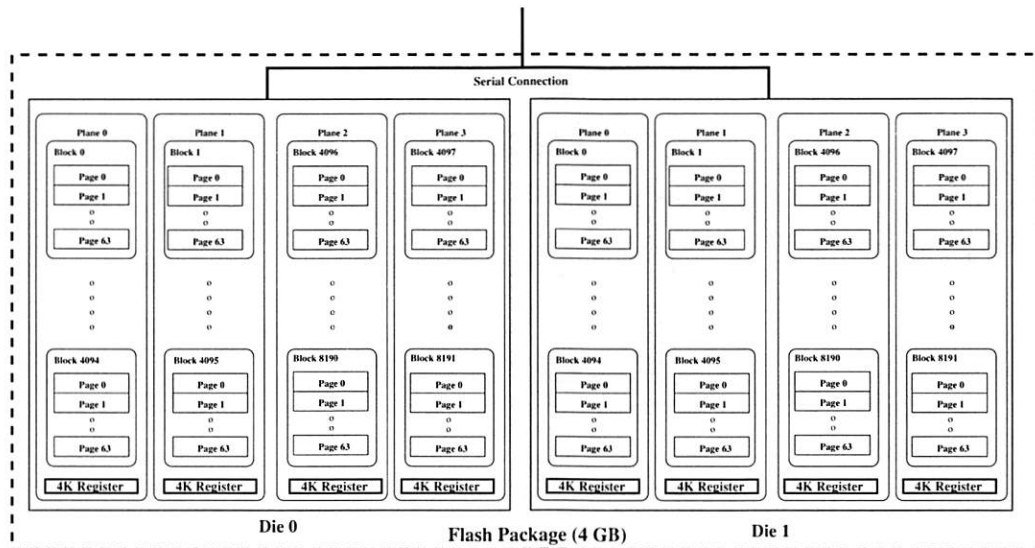


Figure 1: Samsung 4GB flash internals

Interleaving can provide considerable speedups when the operation latency is greater than the serial access latency. For example, a costly erase command can in some cases proceed in parallel with other commands. As another example, fully interleaved page copying between two packages can proceed at close to $100\mu\text{s}$ per page as depicted in Figure 2 in spite of the $200\mu\text{s}$ cost of a single write operation. Here, 4 source planes and 4 destination planes copy pages at speed without performing simultaneous operations on the same plane-pair and while optimally making use of the serial bus pins connected to both flash dies. Once the pipe is loaded, a write completes every interval ($100\mu\text{s}$).

Even when flash architectures support interleaving, they do so with serious constraints. So, for example, operations on the same flash plane cannot be interleaved. This suggests that same-package interleaving is best employed for a choreographed set of related operations, such as a multi-page read or write as depicted in Figure 2. The Samsung parts we examined support a fast internal copy-back operation that allows data to be copied to another block on-chip without crossing the serial pins. This optimization comes at a cost: the data can only be copied within the same flash plane (of 2048 blocks). Two such copies may themselves be interleaved on different planes, and the result yields similar performance to the fully-interleaved inter-package copying depicted in Figure 2, but without monopolizing the serial pins.

3 SSD Basics

In this section we outline some of the basic issues that arise when constructing a solid-state disk from NAND-

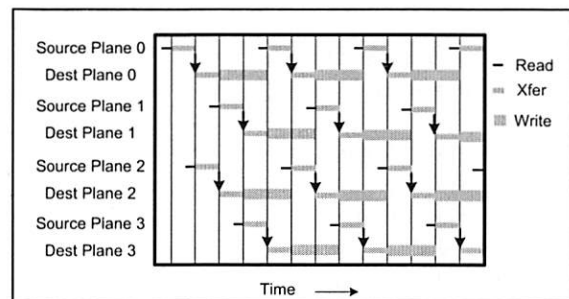


Figure 2: Interleaved page copying

flash components. Although we introduce a number of dimensions in which designs can differ, we leave the evaluation of specific choices until Section 4.

All NAND-based SSDs are constructed from an array of flash packages similar to those described in the previous section. Figure 3 depicts a generalized block diagram for an SSD. Each SSD must contain host interface logic to support some form of physical host interface connection (USB, FiberChannel, PCI Express, SATA) and logical disk emulation, like a *flash translation layer* mechanism to enable the SSD to mimic a hard disk drive. The bandwidth of the host interconnect is often a critical constraint on the performance of the device as a whole, and it must be matched to the performance available to and from the flash array. An internal buffer manager holds pending and satisfied requests along the primary data path. A multiplexer (Flash Demux/Mux) emits commands and handles transport of data along the serial connections to the flash packages. The multiplexer can include additional logic, for example, to buffer commands and data. A processing engine is also required to manage the request flow and mappings from disk logical block

The variables that define allocation pools trade off against these constraints. For example, if a large portion of the LBA space is statically mapped, then there is little scope for load-balancing. If a contiguous range of LBAs is mapped to the same physical die, performance for sequential access in large chunks will suffer. With a small logical page size, more work will be required to eliminate valid pages from erasure candidates. If the logical page size (with unit span) is equal to the block size, then erasure is simplified because the write unit and erase unit are the same, however all writes smaller than the logical page size result in a read-modify-write operation involving the portions of the logical page not being modified.

RAID systems [26] often stripe logically contiguous chunks of data (e.g. 64KB or larger) across multiple physical disks. Here, we use striping at fine granularity to distribute logical pages (4K) across multiple flash dies or packages. Doing so serves both to distribute load and to arrange that consecutive pages will be placed on different packages that can be accessed in parallel.

3.2 Cleaning

Fleshing out the design sketched by Birrell et al. [2], we use flash blocks as the natural allocation unit within an allocation pool. At any given time, a pool can have one or more *active blocks* available to hold incoming writes. To support the continued allocation of fresh active blocks, we need a garbage collector to enumerate previously-used blocks that must be erased and recycled. If the logical page granularity is smaller than the flash block size, then flash blocks must be cleaned prior to erasure. Cleaning can be summarized as follows. When a page write is complete, the previously mapped page location is *superseded* since its contents are now out-of-date. When recycling a candidate block, all non-superseded pages in the candidate must be written elsewhere prior to erasure.

In the worst case, where superseded pages are distributed evenly across all blocks, $N - 1$ cleaning writes must be issued for every new data write (where there are N pages per block). Of course, most workloads produce clusters of write activity, which in turn lead to multiple superseded pages per block when the data is overwritten. We introduce the term *cleaning efficiency* to quantify the ratio of superseded pages to total pages during block cleaning. Although there are many possible algorithms for choosing candidate blocks for recycling, it is always desirable to optimize cleaning efficiency. It's worth noting that the use of striping to enhance parallel access for sequential addresses works against the clustering of superseded pages.

For each allocation pool we maintain a free block list that we populate with recycled blocks. In this section and the next, we assume a purely greedy approach that calls

for choosing blocks to recycle based on potential cleaning efficiency. As described in Section 2, NAND flash sustains only a limited number of erasures per block. Therefore, it is desirable to choose candidates for recycling such that all blocks age evenly. This property is enforced through the process known as wear-leveling. In Section 5, we discuss how the choice of cleaning candidates interacts directly with wear-leveling, and suggest a modified greedy algorithm.

In an SSD that emulates a traditional disk interface, there is no abstraction of a free disk sector. Hence, the SSD is always full with respect to its advertised capacity. In order for cleaning to work, there must be enough spare blocks (not counted in the overall capacity) to allow writes and cleaning to proceed, and to allow for block replacement if a block fails. An SSD can be substantially *overprovisioned* with such spare capacity in order to reduce the demand for cleaning blocks in foreground. Delayed block cleaning might also produce better clustering of superseded pages in non-random workloads.

In the previous subsection, we stipulated that a given LBA is statically mapped to a specific allocation pool. Cleaning can, however, operate at a finer granularity. One reason for doing so is to exploit low-level efficiency in the flash architecture such as the internal copy-back operation described in Section 2.2, which only applies when pages are moved within the same plane. Since a single flash plane of 2048 blocks represents a very small allocation pool for the purposes of load distribution, we would like to allocate from a larger pool. However, if an active block and cleaning state per plane is maintained, then cleaning operations within the same plane can be arranged with high probability.

It might be tempting to view block cleaning as similar to log-cleaning in a Log-Structured File System [28] and indeed there are similarities. However, apart from the obvious difference that we model a block store as opposed to a file system, a log-structured store that writes and cleans in strict disk-order cannot choose candidate blocks so as to yield higher cleaning efficiency. And, as with LFS-like file systems, it's altogether too easy to combine workloads that would cause all recoverable space to be situated far from the log's cleaning pointer. For example, writing the same sets of blocks over and over would require a full cycle over the disk content in order for the cleaning pointer to reach the free space near the end of the log. And, unlike a log-structured file system, the disk here is always "full", corresponding to maximal cleaning pressure all the time.

3.3 Parallelism and Interconnect Density

If an SSD is going to achieve bandwidths or I/O rates greater than the single-chip maxima described in Sec-

tion 2.2, it must be able to handle I/O requests on multiple flash packages in parallel, making use of the additional serial connections to their pins. There are several possible techniques for obtaining such parallelism assuming full connectivity to the flash, some of which we have touched on already.

- **Parallel requests.** In a fully connected flash array, each element is an independent entity and can therefore accept a separate flow of requests. The complexity of the logic necessary to maintain a queue per element, however, may be an issue for implementations with reduced processing capacity.
- **Ganging.** A *gang* of flash packages can be utilized in synchrony to optimize a multi-page request. Doing so can allow multiple packages to be used in parallel without the complexity of multiple queues. However, if only one queue of requests flows to such a gang, elements will lie idle when requests don't span all of them.
- **Interleaving.** As discussed in Section 2.2, interleaving can be used to improve bandwidth and hide the latency of costly operations.
- **Background cleaning.** In a perfect world, cleaning would be performed continuously in the background on otherwise idle components. The use of operations that don't require data to cross the serial interface, such as internal copy-back, can help hide the cost of cleaning.

The situation becomes more interesting when full connectivity to the flash packages is not possible. Two choices are readily apparent for organizing a gang of flash packages: 1) the packages are connected to a serial bus where a controller dynamically selects the target of each command; and 2) each package has separate data path to the controller, but the control pins are connected in a single broadcast bus. Configuration (1) is depicted in Figure 4. Data and control lines are shared, and an enable line for each package selects the target for a command. This scheme increases capacity without requiring more pins, but it does not increase bandwidth. Configuration (2) is depicted in Figure 5. Here there is shared set of control pins, but since there are individual data paths to each package, synchronous operations which span multiple packages can proceed in parallel. The enable lines can be removed from the second configuration, but in this case all operations *must* apply to the entire gang, and no package can lie idle.

Interleaving can play a role within a gang. A long running operation such as block erasure can be performed on one element while reads or writes are proceeding on others (hence the control line need only be held long enough

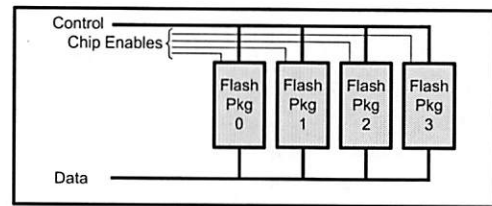


Figure 4: Shared bus gang

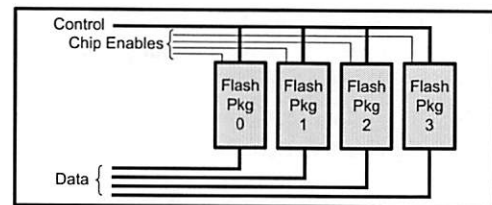


Figure 5: Shared control gang

to issue a command). The only constraint is competition for the shared data and command bus. This could become quite important during block recycling since block erasure is an order of magnitude more expensive than other operations.

In another form of parallelism, intra-plane copy-back can be used to implement block cleaning in background. However, cleaning can take place with somewhat lower latency if pages are streamed at maximum speed between two chips. This benefit comes, of course, at the expense of occupying two sets of controller pins for the duration.

It is unlikely that any one choice for exploiting parallelism can be optimal for all workloads, and certainly as SSD capacity scales up, it will be difficult to ensure full connectivity between controllers and flash packages. The best choices will undoubtedly be dictated by workload properties. For example, a highly sequential workload will benefit from ganging, a workload with inherent parallelism will take advantage of a deeply parallel request queuing structure, and a workload with poor cleaning efficiency (e.g. no locality) will rely on a cleaning strategy that is compatible with the foreground load.

3.4 Persistence

Flash memory is by definition persistent storage. However, in order to recover SSD state, it is essential to rebuild the logical block map and all related data structures. This recovery must also reconstruct knowledge of failed blocks so that they are not re-introduced into active use. There are several possible approaches to this problem. Most take advantage of the fact that each flash page contains a dedicated area (128 bytes) of metadata storage that can be used to store the logical block address that maps to a given flash page. In our simulator, we model the technique sketched by Birrell et al. [2]. This tech-

nique eliminates the need to visit every page at startup by saving mapping information per block rather than per page. Note that in any such algorithm, pages whose content has been superseded but not yet erased will appear multiple times during recovery. In these cases, the stable storage representation must allow the recovery algorithm to determine the most recent instance of a logical page within an allocation pool.

Flash parts do not, in general, provide error detection and correction. These functions must be provided by application firmware. The page metadata can also hold an error-detection code to determine which pages are valid and which blocks are failure-free, and an error-correction code to recover from the single-bit errors that are expected with NAND flash. Samsung specifies block lifetime assuming the presence of a single-bit ECC [29]. It may be possible to extend block lifetime by using more robust error correction.

The problem of recovering SSD state can be bypassed altogether by holding the logical block map in Phase-Change RAM [14] or Magnetoresistive RAM [9]. These non-volatile memories are writable at byte granularity and don't have the block-erase constraints of NAND flash. The former is not yet widely available, and the latter can be obtained in small capacities, but at 1000 times the cost of NAND flash. Alternatively, backup power (e.g. big capacitors) might be enough to flush the necessary recovery state to flash on demand.

3.5 Industry Trends

The market for NAND flash continues to grow both in volume and in breadth as storage capacity increases and cost per unit storage declines. As of late 2007, laptops are available that use an SSD as the primary disk drive. In addition, high-end flash systems have become available for the enterprise marketplace. Most products in the marketplace can be placed into one of three categories.

Consumer portable storage. These are inexpensive units with one or perhaps two flash packages and a simple controller. They commonly appear as USB flash sticks or camera memories, and feature moderate bandwidth for sequential operations, moderate random read performance, and very poor random write performance.

Laptop disk replacements. These disks provide substantial bandwidth to approximate that of the IDE and SATA disks they replace. Random read performance is far superior to that of rotating media. Random write performance is comparable to that of rotating media.

Enterprise/database accelerators. These units promise very fast sequential performance, random read performance superior to that of a high-end RAID array, and very strong random write performance. They have costs to match. However, the specified random-write perfor-

	Sequential		Random 4K	
	Read	Write	Read	Write
USB	11.7 MB/sec	4.3 MB/sec	150/sec	<20/sec
MTron	100 MB/sec	80 MB/sec	11K/sec	130/sec
Zeus	200 MB/sec	100 MB/sec	52K/sec	11K/sec
FusionIO	700 MB/sec	600 MB/sec	87K/sec	Not avail

Table 2: Sample Flash Device Performance

mance numbers are often a bit mysterious, and sometimes are missing altogether. For example, it is often difficult to tell whether cleaning costs are included.

In Table 2, we provide performance numbers for sample devices in these categories. The sample devices are the Lexar JD Firefly 8GB USB flash stick, the MTron MSD-SATA3025 SSD [20], the Zeus^{TOPS} SSD [30], and the FusionIO ioDrive [10]. The MTron device is in the second category above, while the Zeus and FusionIO devices are clearly enterprise-class. We are not privy to the internals of these devices, but we can speculate on how they are organized. Previous work [2] points out that consumer USB flash devices almost certainly use a logical page granularity close to the size of a flash block. Limited random read performance suggests that only a single I/O request is in flight at a time. The MTron device gives random read I/O performance better than can be expected with a single flash request stream, so it almost certainly employs some sort of parallel request structure. However, the poor random write I/O performance suggests that a large logical page size is being used (with substantial read-modify-write costs). The random write performance of the Zeus and FusionIO devices suggest a sophisticated block mapping scheme, and multiple parallel activities are clearly needed to reach the stated random I/O performance. The Zeus system appears to lower pressure on its cleaning algorithm by substantial over-provisioning [7]. The FusionIO system (for which only a preliminary specification was available in 2007) is the first to offer a PCI-Express interconnect.

4 Design Details and Evaluation

From our discussion of industry trends, it is clear that current SSDs can get very good performance numbers from ideal workloads. For example, under sequential workloads these devices verge on saturating any interconnect, RAID controller [24], or host peripheral chipset [1]. However, it's unclear whether this is true of a single sequential stream, or multiple streams. And if multiple, how many streams are optimal? Does the stated write performance account for cleaning overhead? What is the random write performance and what are the assumptions about distribution of superseded pages during cleaning?

This section introduces a trace-driven simulation environment that allows us to gain insight into SSD behavior under various workloads.

4.1 Simulator

Our simulation environment is a modified version of the DiskSim simulator [4] from the CMU Parallel Data Lab. DiskSim does not specifically support simulation of solid-state disks, but its infrastructure for processing trace logs and its extensibility made it a good vehicle for customization.

DiskSim emulates a hierarchy of storage components such as buses and controllers (e.g. RAID arrays) as well as disks. We implemented an SSD module derived from the generic rotating disk module. Since this module did not originally support multiple request queues, we added an auxiliary level of parallel elements, each with a closed queue, to represent flash elements or gangs. We also added logic to serialize request completions from these parallel elements. For each element, we maintain data structures to represent SSD logical block maps, cleaning state, and wear-leveling state. As each request is processed, sufficient delay is introduced to simulate real-time delay according to the specifications in Table 1. If cleaning and recycling is called for by the simulator state, additional delay is introduced to account for it, and the state is updated accordingly. We added configuration parameters to enable such features as background cleaning, gang-size, gang organization (e.g. switched or shared-control), interleaving, and overprovisioning.

Verifying our simulation requires detailed experiments to determine the caching and flash-management algorithms used by actual SSD hardware. We intend to do this as future work.

4.2 Workloads

We present results for a collection of workload traces which we name as follows for the purpose of exposition: TPC-C, Exchange, IOzone, and Postmark.

We first examine a synthetic workload generated by DiskSim. We present this workload to characterize baseline behavior for sequential and random access request streams. IOzone [15] and Postmark [16] are standard file system benchmarks run on a workstation class PC with a 750 GB SATA disk. These benchmarks require relatively little capacity, and can be simulated on a single SSD. Despite the fact that we do not simulate on-disk caching, in the above traces, the disk cache was enabled, producing unnaturally low request inter-arrival times for writes.

TPC-C is an instance of the well-established database benchmark [31]. Our trace is a 30-minute trace of a

large-scale TPC-C configuration, running 16,000 warehouses. The traced system comprised 14 RAID (HP MSA1500 Fibre-Channel) controllers each supporting 28 high-speed 36 GB disks. We target one of the controllers serving non-log data tables: a mixed read/write workload with about twice as many reads as writes. (The 13 non-log controllers have similar workloads.) Although each controller manages over a terabyte of storage, the benchmark uses only about 160GB per controller. The large number of disks are needed to obtain disk arms that can handle requests in parallel. All requests in this workload are for multiples of 8KB blocks. Alignment is important, since misaligned requests to flash add a page access to every read or write. Several of the logical volumes in our configuration were misaligned, yielding traces in which $LBA \bmod 8 = 7$ for all LBA . We corrected for this by post-processing the roughly 6.8M events in this trace.

The Exchange workload is taken from a server running Microsoft Exchange. This is a specialized database workload with about a 3:2 read-to-write ratio. The traced server had 6 non-log RAID controllers of a terabyte each (14 disks). We extracted a 15 minute trace of roughly 65000 events from one of these controllers, involving requests over 250GB of disk capacity.

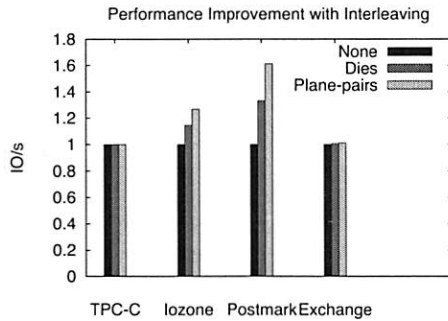
4.3 Simulation Results

We first present results from a simple workload synthesized by the DiskSim workload-generator. Then, we vary different configuration parameters and study their impact on SSD performance under the macro benchmarks.

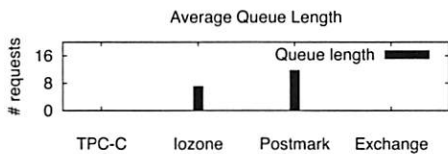
Our baseline configuration is an SSD with 32GB of flash: 8 fully connected flash packages. In this configuration, allocation pools are the size of a flash package, the logical page and stripe size is 4KB, and cleaning requires data transfer across the flash package serial interface. Because we model only a small SSD, the larger workloads require that we simulate a RAID controller as well. We assume that each SSD is overprovisioned by 15%, which means that the disk capacity available to the host is around 27 GB. We invoke cleaning when less than 5% free blocks remain. The TPC-C workload requires 6 attached SSDs in this configuration, and the Exchange workload requires 10.

Microbenchmark	Cleaning	Latency (μ s)	IO/s
Sequential read	x	130	61,255
Random read	x	130	61,255
Sequential write	x	309	25,898
Random write	x	309	25,898
Sequential write	✓	327	24,457
Random write	✓	433	18,480

Table 3: SSD performance under microbenchmarks



(a)



(b)

Figure 6: Impact of interleaving

Microbenchmarks. We ran a set of 6 synthetic microbenchmarks involving 4 KB I/O operations and report the access latency and respective I/O rates in Table 3. In a fully-connected SSD, sequential and random I/Os have equivalent latencies. Note that this latency includes the time to transfer both the page data and the 128-byte page metadata. When cleaning is enabled, the latencies for write operations reflect the additional overhead. Notice that sequential writes result in better cleaning efficiency, and therefore less cleaning overhead.

Page Size, Striping, and Interleaving. Choice of logical page size has a substantial impact on overall performance. As discussed in Section 3.1, every write that is smaller than the logical page size requires a read-modify-write operation. When run with a full-block page size (256KB) at unit depth (e.g., the entire logical page on the same die), TPC-C produces an average I/O latency of over 20 ms, more than two orders of magnitude greater than what can be expected with a 4KB page size. Our eight package configuration with a 256KB page size can (barely) keep up with the average trace rate of 300 IOPS per SSD, but only due to the inherent parallelism available in the SSD. We do much better with a smaller page size. The average latency for TPC-C is 200 μ s with a page size of 4KB, although the workload does not have enough events to test the 40,000 IOPS that this implies. As described in Section 2.2, I/O performance can be improved by interleaving multiple requests within a single flash package or die. Our simulator accounts for interleaving by noticing when two requests are queued on a flash package that can proceed concurrently according to the hardware constraints. Figure 6(a) presents I/O rates

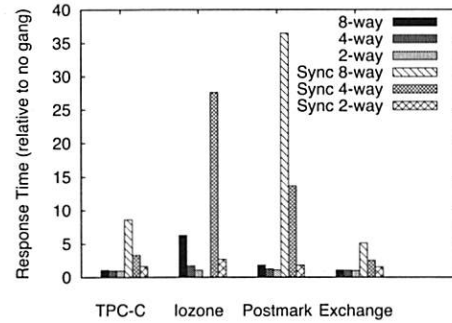


Figure 7: Shared-control ganging

normalized with respect to our baseline configuration and shows how various types of interleaving improves the performance of our baseline configuration. While I/Ozone and Postmark show an increase in throughput, TPC-C and Exchange do not benefit from interleaving. As shown in Figure 6(b), the average number of queued requests (per flash package, as measured by DiskSim) is very close to zero for these two workloads. With no queuing, interleaving will not occur. I/Ozone and Postmark have a significant sequential I/O component. When a large sequential request is dispatched to multiple packages due to stripe boundaries, queuing occurs and interleaving becomes beneficial. One might think that TPC-C would benefit from striping its 8KB requests at 8KB increments thereby allowing every request to interleave at the package or die level. However, splitting up each request into parallel 4KB requests is in this case superior.

	No gang	8-gang	16-gang
Host IO Latency	237 μ s	553 μ s	746 μ s
IOPS per gang	4425	1807	1340

Table 4: Shared-bus gang performance for Exchange

Gang Performance. As suggested in Section 3.3, ganging flash components offers the possibility of scaling capacity without linearly scaling pin density and firmware logic complexity. We proposed two types of ganging: shared-bus and shared-control. Table 4 shows the average latency of Exchange I/O requests (variable size) under 8-wide (32KB) and 16-wide (64KB) shared-bus gangs. As it happens, this workload requires only about 900 IOPS, so the 16-gang is fast enough even though the ganged components have to be accessed serially. There is no obvious load-balancing problem when simple page-level striping is used, even though one would expect such problems to be exacerbated by ganging.

A shared-control gang can be organized in two ways. First, although the flash packages are ganged, separate allocation and cleaning decisions can be made on each

package, enabling one to perform opportunistic parallel operations, e.g., when two reads are presented on different gang members at the same time, they can be performed concurrently. We refer to this as asynchronous-shared-control ganging. Second, all packages in a gang can be managed in synchrony by utilizing a logical page depth equal to that of gang size, e.g., a 8-wide gang would have a page size of 32KB, and we call this design synchronous-shared-control ganging. We use intra-plane copy-back to implement read-modify-write for writes of less than a page in synchronous ganging.

Figure 7 presents normalized response time (with respect to the base line configuration) from various synchronous and asynchronous shared-control gang sizes. Since the logical page size of a synchronous gang is bigger than the corresponding asynchronous gang, it limits the number of simultaneous operations that can be performed in a gang unit, and hence synchronous ganging uniformly underperforms when compared to asynchronous ganging. The synchronous 8-way gang could not support the IOzone workload in simulated real-time and hence its result is absent in the Figure 7.

	# cleaned	Avg. time (ms)	Efficiency
TPC-C (inter-plane)	114	9.65	70%
TPC-C (copy-back)	108	5.85	70%
IOzone	101170	1.5	100%
Postmark	2693	1.5	100%

Table 5: Cleaning frequency and efficiency

Copy-back vs. Inter-plane Transfer. Cleaning a block involves moving any valid pages to another block. If the source and destination blocks are within a plane, pages can be moved using the copy-back feature without having to transfer them across the serial pins. Otherwise, pages can be moved between planes through the serial pins. Table 5 presents the average number of blocks cleaned per flash package, the average time to clean a block, and the average cleaning efficiency. Using the copy-back feature, TPC-C shows a 40% improvement in cleaning cost per block. In spite of the large number blocks being cleaned, IOzone and Postmark do not show any benefit from copy-back. These benchmarks produce perfect cleaning efficiency; they move no pages during cleaning.

Cleaning Thresholds. An SSD needs a minimum number of free blocks to operate correctly; for example, free blocks are required to perform data transfer during cleaning or to sustain sudden bursts of write requests. Increasing this minimum-block threshold triggers cleaning earlier and therefore increases observed overhead. Figure 8(a) shows the variation in access latency as we increase the free blocks threshold. While the access laten-

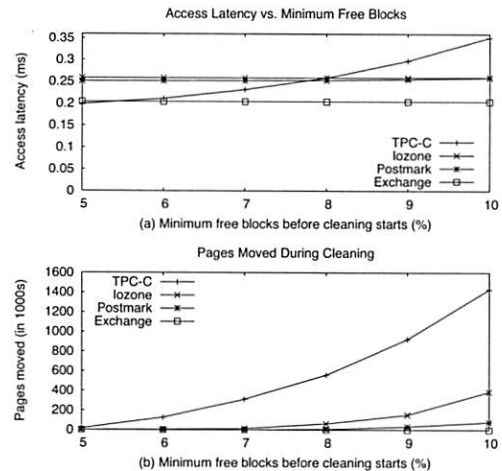


Figure 8: Impact of minimum free blocks

cies increase in TPC-C with the threshold, other workloads show little difference. This difference in the access latencies among different workloads is explained by Figure 8(b), which plots the number of pages moved during cleaning against the free blocks threshold. Figures 8(a) and 8(b) show that increasing the minimum free blocks threshold may affect the overall performance of the SSD depending upon the pages moved under the workload.

4.4 Tradeoff Summary

In Table 6, we present a brief summary of the benefits and drawbacks of the design techniques discussed above. We believe that these tradeoffs are largely independent of each other, but leave a rigorous examination of this hypothesis for future work.

Technique	Positives	Negatives
Large allocation pool	Load balancing	Few intra-chip ops
Large page size	Small page table	Read-modify-writes
Overprovisioning	Less cleaning	Reduced capacity
Ganging	Sparser wiring	Reduced parallelism
Stripping	Concurrency	Loss of locality

Table 6: SSD Design Tradeoffs in Brief

5 Wear-leveling

In the discussion below, we propose a cleaning and wear-leveling algorithm applicable to NAND-flash SSDs. We assume that an SSD implements a block-oriented disk interface which provides no *a priori* knowledge of optimal data placement or likely longevity.

Efficient cleaning, while it may reduce overall wear, does not translate to even wear. The drawback of choosing a greedy approach (maximal cleaning efficiency) is that the same blocks may get used over and over again

and a large collection of blocks with relatively cold content may remain unused. For example, if 50% of the blocks contain cold data that is never superseded, and the rest contains hot data that is modified frequently, then the block to be erased will always be taken from the hot blocks. This will lead to a situation where the life of the hot blocks will be consumed while that of the cold blocks will be unutilized, wasting half the total life of the device.

Our objective is to design a block management algorithm so as to delay the expiry time of any single block; that is, we wish to avoid the situation where one or a few blocks have finished their life when most blocks have much life left. This goal implies that we must ensure that the remaining lifetime across all blocks is equal within an acceptable variance. To this end, we propose tracking the average lifetime remaining over all blocks. The remaining lifetime of any block should be within *ageVariance* (say 20%) of the average remaining lifetime.

This desired policy can be achieved by running the greedy strategy as long as it picks a page whose remaining lifetime is above the threshold. To do this, we must maintain some notion of block erase count in persistent storage (for example in the metadata portion of the first page). What then should we do for *worn out* blocks that drop below the threshold? A simple approach is to only allow recycling when a candidate's remaining lifetime exceeds the threshold. Doing this could exclude large numbers of blocks from consideration which in turn would cause the remaining blocks to be recycled more frequently and with poorer cleaning efficiency. For example if 25% of the blocks have cold data and the remaining 75% have hot data accessed uniformly then after a certain number of writes, the latter will get worn out and become ineligible for erasure. Subsequently, recycling will be concentrated on the 25% of the blocks containing cold data. So these blocks will be reused 4 times faster and yield commensurately fewer pages per erasure. Hence, we need to avoid making a large number of blocks ineligible for recycling over an extended period of time.

Instead of freezing the recycling of worn out blocks, we can rate-limit their usage. Randomization can be used here to evenly spread out the effect of a rate-limit on the worn out blocks. We use an approach similar to Random Early Discard [8] in which the probability of recycling drops linearly from 1 to 0 as a block's remaining lifetime drops from say 80% to 0% of the average.

Another way to slow down the usage of worn out blocks is to migrate cold data into old blocks. When data is migrated, cleaning is performed as usual, but then rather than attaching the recycled block to the allocation pool queue, instead data from a cold block is used to fill it. The cold block is then recycled and added to the free queue. This action can be triggered,

for example, if the remaining lifetime in a block drops below *retirementAge* (say 85% of the average remaining lifetime). *retirementAge* should be less than *ageVariance* of average remaining lifetime so that cold data can be migrated into a worn out block before rate-limiting kicks in.

One method to identify cold data is to look for blocks that have exceeded specified parameters for remaining lifetime and time since last erasure. This approximation can be made more accurate by keeping track of when a block was last written in its metadata. In this case, it is important that temperature metadata travel with the content as it is moved to a new physical block. When a block is migrated, the migration should not affect its temperature metric. However, the process of cleaning can group pages of different temperatures in the same block, in this case, the resultant block temperature needs to reflect that of the aggregate.

So in summary, we propose running the greedy strategy (e.g., the most superseded pages) for picking the next block to be recycled, as modified below.

- If the remaining lifetime in the chosen block is below *retirementAge* of the average remaining lifetime then migrate cold data into this block from a migration-candidate queue, and recycle the head block of the queue. Populate the queue with blocks exceeding parametric thresholds for remaining lifetime and duration, or alternatively, choose migration candidates by tracking content temperature.
- Otherwise, if the remaining lifetime in the chosen block is below *ageVariance*, then restrict recycling of the block with a probability that increases linearly as the remaining lifetime drops to 0.

5.1 Wear-leveling Simulation

We ran IOzone (due to its high cleaning rate) to study the wear-leveling algorithm described above. We reduced the lifetime of a flash block from 100K to 50 cycles for our experiment so that the *ageVariance* (set to 20%) and *retirementAge* (set to 85%) thresholds become relevant. Tables 7 and 8 present the results for 3 different techniques: the greedy algorithm, greedy with rate-limited cleaning of worn-out blocks, and greedy with rate-limiting and cold data migration. Although the average block lifetime is similar across the techniques, invoking migration gives a much smaller standard deviation of remaining block lifetimes at the end of the run. Moreover, with migration, there were no block expiries (e.g., blocks over the erasure limit). Since the simple greedy technique does not perform any rate-limiting, fewer blocks reach expiry when rate-limiting is used than without it. Table 8 presents the distribution of flash-block

lifetimes around the mean. One can observe that cold data migration offers better clustering around the mean than the other options.

	Mean Lifetime	Std.Dev.	Expired blocks
Greedy	43.82	13.47	223
+ Rate-limiting	43.82	13.42	153
+ Migration	43.34	5.71	0

Table 7: Block wear in IOzone

	< 40%	< 80%	< 100%	≥ 100%
Greedy	1442	1323	523	13096
+ Rate-limiting	1449	1341	501	13092
+ Migration	0	0	8987	7397

Table 8: Lifetime distribution with respect to mean

The cost of migrating cold pages across blocks imposes a performance cost that is workload-dependent. Our simulation of wear-leveling for IOzone involved 7902 migrations per package which added a 4.7% overhead to the average I/O operation latency.

5.2 Opening the Box

A system that implements a traditional disk-block interface incurs unnecessary overhead by managing disk blocks that are free from the point of view of the file system. Under a random workload, a disk that is half full will have twice the cleaning efficiency of a full disk (and all disks are full except those that are overprovisioned). Previous work on Semantically-Smart Disk Systems [21] has shown the benefits of greater file-system information being available at the disk level. Although SSDs that implement a pure disk interface provide advantages from a perspective of compatibility, it is worth considering whether the SSD API might support the abstraction of an unused block. With such a modification, SSD performance would vary with the percentage of free space rather than always suffering maximal cleaning load and wear.

Cleaning load can be reduced if an SSD has knowledge of content volatility. For example, certain file types such as audio and video files are not often modified. If available at the disk block level, this information would provide a better predictive metric than the history-based approximations above. More importantly, if cold data can be identified *a priori*, then there is a better chance of establishing locality for warm data, localization of warm data will lead to better cleaning efficiency.

6 Related Work

We discuss related work in designing solid-state storage devices, file systems for improving performance, and work on algorithms and data structures for such devices.

6.1 Solid-State Storage Devices

Previous work on solid-state storage design has focused on resource-constrained environments such as embedded systems or sensor networks (e.g., Capsule [19], MicroHash [34]). This body of work has largely dealt with small flash devices (up to a few hundred MB), with low-power, shock resistance and size being primary considerations. The MicroHash index [34] attempts to support temporal queries on data stored locally on a flash chip in the presence of a low energy budget. Nath and Kansal propose FlashDB [23], a hybrid B+-tree index design. The key idea is to have different update strategies depending on the frequency of reads and writes: in-place updates for pages that are frequently read or infrequently written, and logging for those that are frequently written.

While the work in embedded and sensor environments has given useful insights into the workings and constraints of solid-state devices, our work systematically explores design issues in high-performance storage systems. In these environments, operation throughput is often the most important metric of interest.

Hybrid disks are another area of research [3] and commercial interest. These devices place a small amount of flash memory alongside a much larger traditional disk to improve performance. Flash is not the final persistent store, but rather a write-cache to improve latency. The non-volatile cache on hybrid disks can be controlled through specific ATA commands [25].

File systems have also used non-volatile memory to log data or requests. WAFL [13] is one such file system that uses non-volatile RAM (NVRAM) to keep a log of NFS requests it has processed since the last consistency point. After an unclean shutdown, WAFL replays any requests in the log to prevent them from being lost.

The hybrid disk and NVRAM approaches use flash as an add-on storage for rotating disks. In our designs, solid-state devices serve as a replacement for rotating disks, providing a better rate of operation throughput.

Kim and Ahn [17] propose a cache-management strategy that improves random-write performance for SSDs operating with a block-sized logical page. They attempt to flush write-cache pages that occupy the same block at the same time, thereby reducing read-modify-write overhead. This works well if the workload does not overwhelm the cache or require immediate write persistence. Moreover, write-caching that handles bursty or repetitive writes is complementary to our approach.

6.2 File System Designs

File systems specific to flash devices have also been proposed. Most of these designs are based on Log-structured File Systems [28], as a way to compensate

for the write latency associated with erasures. JFFS, and its successor JFFS2 [27], are journaling file systems for flash. The JFFS file systems are not memory-efficient for storing volatile data structures, and require a full scan to reconstruct these data structures from persistent storage upon a crash. JFFS2 performs wear-leveling, in a somewhat ad-hoc fashion, with the cleaner selecting a block with valid data at every 100th cleaning, and one with most invalid data at other times.

YAFFS [18] is flash file system for use in embedded devices. It treats handling of wear-leveling akin to handling bad blocks, which appear as the device gets used. Other examples of embedded micro-controller file systems include the Transactional Flash File System [11] and the Efficient Log Structured Flash File System [6]. The former was designed for more expensive, byte addressable NOR flash memory, which has considerably fewer constraints than NAND flash. The latter was designed for sensor nodes using NAND flash. It supports simple garbage collection and provides an optional best effort crash recovery mechanism.

It is useful to compare our approach with improvements higher up in the storage stack, such as the specialized file systems for flash devices. Enhancements at the flash controller will obviate the need to invest significant effort in re-writing a custom flash file system. It will also alleviate the overhead of transitioning from rotating disks to flash-based storage by exporting a “flash-disk” that performs well even with existing file systems.

6.3 Algorithms and Data-structures

Much prior work has been done on proposing and evaluating algorithms and data structures specially suited for operation in flash devices. A recent survey [12] discusses much of this work in greater detail.

Wear-leveling is an important constraint of flash devices and several proposals have been made to perform it efficiently, increasing the usable life time of the device. Wu and Zwaenepoel [33] use a relative wear-count of blocks for wear-leveling. Similar to our approach, data is swapped when the block chosen for cleaning exceeds a wear-count. Wells [32] proposed a reclamation policy based on weighted combination of efficiency and wear-leveling, while the work by Chiang and Chang [5] uses the likelihood of a block being used soon, which is equivalent to the logical hotness or coldness of data within the block chosen for cleaning.

Recent work by Myers [22] looks at ways to exploit the inherent parallelism offered by the flash chip. He fragments a block and stores it on multiple physical pages on different chips, under the hypotheses that a dynamic striping or replication strategy based on workload will outperform a static one. His work focuses on ap-

plicability of flash for database workloads and concludes that widespread adoption is not yet possible. In contrast, our design and analysis shows that while there are several tradeoffs, SSDs are a viable and possibly an attractive option for transactional workloads such as TPC-C.

7 Conclusion

As we have shown, there are numerous design tradeoffs for SSDs that impact performance. There is also significant interplay between both the hardware and software components and the workload. Our work provides insight into how all of these components must cooperate in order to produce an SSD design that meets the performance goals of the targeted workload. From the hardware standpoint, the SSD interface (SATA, IDE, PCI-Express) and package organization dictate theoretical maximum I/O performance. On the software side, the properties of the allocation pool, load balancing, data placement, and block management (wear-leveling and cleaning) combined with workload characteristics determine overall SSD performance. Moreover, we have demonstrated that all designs can benefit from plane interleaving and some degree of overprovisioning, and we have proposed a wear-leveling algorithm and shown its efficacy in at least one scenario.

We have demonstrated a simulation-based technique for modeling SSD performance driven by traces extracted from real hardware. In some cases, the traced systems require storage components that would be much too expensive for most organizations to provide for the purposes of experimentation. Our simulation framework has proved both resilient and flexible, and we expect to continue to add to the set of behaviors that we can model. Shared-control ganging and refined wear-leveling data are particular topics of interest.

There is no fixed rule that NAND flash be integrated into computer systems as disk storage. However, the block-access nature of NAND suggests that a block-oriented interface will often be appropriate. Although outside of the scope of this work, we suspect that our simulation techniques will be applicable to NAND-flash block-storage independent of architecture because the same issues (e.g. cleaning, wear-leveling) will still arise.

Flash-based storage is certain to play an important role in future storage architectures. One corollary of our simulation results is that the storage systems necessary to support a substantial TPC-C workload, which in the past have involved many hundreds of spindles, might well be replaced in future by small numbers of SSD-like devices. Our work represents a step towards understanding and optimizing the performance of such systems.

Acknowledgements

We are grateful to the DiskSim team for making their simulator available, and to Dushyanth Narayanan for porting it to Windows. Andrew Birrell, Chuck Thacker, and James Hamilton participated in many fruitful discussions during the course of this work. Bruce Worthington and Swaroop Kavalanekar gathered our TPC-C and Exchange traces. And finally, we would like to honor the memory of Dr. Jim Gray, who inspired this work.

References

- [1] AnandTech. MTRON SSD 32GB: Wile E. Coyote or Road Runner? <http://www.anandtech.com/storage/showdoc.aspx?i=3064>.
- [2] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A Design for High-Performance Flash Disks. *Operating Systems Review*, 41(2):88–93, 2007.
- [3] T. Bisson and S. A. Brandt. Reducing Hybrid Disk Write Latency with Flash-Backed I/O Requests. In *MASCOTS '07: Proceedings of the 15th IEEE International Symposium on Modeling, Analysis, and Simulation*, 2007.
- [4] J. S. Bucy, G. R. Ganger, and et al. The DiskSim Simulation Environment Version 3.0 Reference Manual. <http://citeseer.ist.psu.edu/bucy03disksim.html>.
- [5] M.-L. Chiang and R.-C. Chang. Cleaning Policies in Mobile Computers Using Flash Memory. *Journal of Systems and Software*, 48(3):213–231, 1999.
- [6] H. Dai, M. Neufeld, and R. Han. ELF: An Efficient Log-Structured Flash File System for Micro Sensor Nodes. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 176–187, 2004.
- [7] D. Dumitru. Understanding Flash SSD Performance. <http://managedflash.com/news/papers/easyco-flashperformance-art.pdf>.
- [8] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.
- [9] Freescale Semiconductor. 256K x 16-Bit 3.3-V Asynchronous Magnetoresistive RAM. http://www.freescale.com/files/microcontrollers/doc/data_sheet/MR2A16A.pdf.
- [10] FusionIO Corporation. ioDrive Datasheet. <http://www.fusionio.com/iodrivedata.pdf>.
- [11] E. Gal and S. Toledo. A Transactional Flash File System for Microcontrollers. In *Proceedings of the USENIX Annual Technical Conference*, pages 89–104, 2005.
- [12] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37(2):138–163, 2005.
- [13] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, 1994.
- [14] IBM Corporation. Promising New Memory Chip Technology Demonstrated By IBM, Macronix & Qimonda Joint Research Team. http://domino.research.ibm.com/comm/pr.nsf/pages/news.20061211_phasechange.html.
- [15] IOzone.org. IOzone Filesystem Benchmark. <http://www.iozone.org>.
- [16] J. Katcher. PostMark: a New File System Benchmark. Technical Report TR3022, Network Appliance, October 1997.
- [17] H. Kim and S. Ahn. A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 239–252, 2008.
- [18] C. Manning. YAFFS: Yet Another Flash File System. <http://www.aleph1.co.uk/yaffs>, 2004.
- [19] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: An Energy-Optimized Object Storage System for Memory-Constrained Sensor Devices. In *SenSys '06: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 195–208, 2006.
- [20] MTron Co., Ltd. MSD-SATA3025 Product Specification. http://mtron.net/Upload_Data/Spec/ASiC/MSD-SATA3025.pdf.
- [21] Muthian Sivathanu and Vijayan Prabhakaran and Florentina I. Popovici and Timothy E. Denehy and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, 2003.
- [22] D. Myers. On the Use of NAND Flash Memory in High-Performance Relational Databases. Master's thesis, MIT, 2007.
- [23] S. Nath and A. Kansal. FlashDB: Dynamic Self-Tuning Database for NAND Flash. In *IPSN '07: Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, pages 410–419, 2007.
- [24] Next Level Hardware. Battleship MTron. <http://www.nextlevelhardware.com/storage/battleship/>.
- [25] N. Obr and F. Shu. A Non-Volatile Cache Command Proposal for ATA8-ACS. <http://t13.org>, 2005.
- [26] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, pages 109–116, 1988.
- [27] Red Hat Corporation. JFFS2: The Jjournaling Flash File System. <http://sources.redhat.com/jffs2/jffs2.pdf>, 2001.
- [28] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.
- [29] Samsung Corporation. K9XXG08XXM Flash Memory Specification. http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/NANDFlash/SLC_LargeBlock/8Gbit/K9F8G08U0M/ds_k9f8g08x0m_rev10.pdf, 2007.
- [30] STEC Incorporated. Zeus^{IOPS} Solid State Drive. http://www.stec-inc.com/downloads/flash_datasheets/iopsdatasheet.pdf.
- [31] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification. http://www.tpc.org/tpcc/spec/tpcc_current.pdf.
- [32] S. E. Wells. Method for Wear Leveling in a Flash EEPROM Memory. US patent 5,341,339, Aug 1994.
- [33] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *ASPLOS-VI: Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–97, 1994.
- [34] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An Efficient Index Structure for Flash-Based Sensor Devices. In *FAST'05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 31–44, 2005.

Context-Aware Mechanisms for Reducing Interactive Delays of Energy Management in Disks

Igor Crk Chris Gniady
University of Arizona
{icrk, gniady}@cs.arizona.edu

Abstract

Aggressive energy conserving mechanisms can maximize energy efficiency, but often have the negative trade-off of simultaneously reducing system responsiveness due to the switching of component power modes. This side-effect is especially prominent in hard disk drives, where the time required to switch power modes is dictated by the latency of the mechanical elements of the drive. Existing disk activity prediction schemes provide solutions for eliminating transition delays in the presence of non-interactive applications and processes, but perform poorly on systems dominated by interactive applications. The key idea in eliminating transition delays exposed to users in interactive applications is that the users are responsible for placing energy and performance demand on the systems through interactions with applications. Therefore, monitoring user interactions with applications provides an opportunity for predicting upcoming power mode transitions and, as a result, eliminating the delays associated with these transitions. In this paper, we propose a set of user behavior monitoring and prediction mechanisms that significantly reduce delays in interactive applications while minimizing energy consumption.

1 Introduction

Energy is a critical system resource for both portable and stationary systems. The need for energy efficiency in portable systems is clear: batteries have a limited energy capacity and users are expecting not only higher performance but longer battery life with every new system they buy. Recently, researchers have realized the positive financial and environmental implications of energy conservation for stand-alone servers and server clusters [3, 4, 12, 22]. The challenge of designing energy efficient systems lies in understanding the role of user interactions in energy consumption and in providing an energy/performance schedule that accommodates user de-

mand. Furthermore, by understanding user behavior we can optimize system performance by tailoring it to a user's patterns of interaction.

Performance and energy consumption are tightly coupled where higher performance is usually achieved at the cost of increased power demand. Likewise, decreasing energy consumption by decreasing the performance level of a component can significantly increase interactive delays. This is particularly apparent in the case of hard disk drives, where the retrieval of data from a spun-down disk results in a significant delay when platters are spun up to operational speed and during which the system may become unresponsive. Keeping the disk spinning and ready to serve requests eliminates interactive delays, but wastes energy. Stopping or slowing the rotation of disk platters during periods of idleness, i.e. periods during which I/O requests are absent, is the most effective means of reducing the energy consumed by a hard drive. While prior research has focused on predicting the upcoming idle periods in order to place the disk in a lower power mode. Little has been done in predicting the arrival of I/O activity, especially in the arena of interactive user applications, where user-generated I/O requests alone do not generally exhibit discernible patterns.

Timeliness of power mode transitions affects not only the system's performance and the overall system's energy consumption but also user perception of the system's responsiveness. Significant delays are associated with the transition to a higher performance state. For example, waiting for I/O requests to arrive before switching to a higher power level may degrade system performance, keep the system processing the task longer and as result increase overall energy consumption. Switching too early wastes energy, since the demand for high performance is not present. Therefore, timely transitions to the appropriate performance level are critical for achieving both best performance and energy efficiency. Monitoring user behavior provides not only the necessary context of execution that was previously unavailable to the

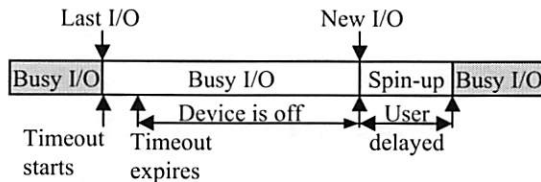


Figure 1: Anatomy of an idle period.

predictors, but enables timely predictions before the need for high performance arrives [8, 1].

In this paper, we show that user interactions can be easily monitored and exploited to increase both the timeliness and accuracy of prediction mechanisms. More specifically, we propose and apply the Interaction-Aware Spin-up Predictor (IASP) to reducing the interactive delays of hard disk power management. We propose a set of mechanisms for capturing user actions and predicting the upcoming device state, and provide a detailed design and implementation. The proposed mechanisms gather contextual information from user's mouse interactions within a GUI and use it in predicting an upcoming I/O request. The idea is motivated by the observation that with a majority of common interactive applications, the user fully interacts with the application through its graphical user interface (GUI). In this context, a simple action such as opening a file requires a sequence of mouse events. By correlating the sequence of steps to the resulting I/O we can predict future I/O occurrences when the user initiates the same set of operations again.

In this paper, we make following set of contributions: (1) we are the first to apply interaction aware prediction to spin up a hard drive, (2) we are able to successfully apply our mechanisms to predicting disk spin-ups in interactive applications, (3) we design, implement, and evaluate our design, showing significant improvements in delays exposed to the user, (4) we extend ALT mechanisms to predict the length of idle periods and spin-up the disk accordingly. Furthermore, the interaction-aware approach can easily be extended to manage other system resources and peripheral devices whose activity is dependent on user behavior.

2 Background

High performance hard drives are a significant source of energy consumption and timeout mechanisms have gained wide popularity due to the simplicity of implementation and the energy savings they provide to disks that would otherwise be spinning needlessly. Figure 1 shows an example of a timeout mechanism shutting down the device once a timer expires. The disk remains powered down until a new I/O request arrives and the

disk has to be powered up before servicing the new request, potentially exposing several seconds of delay to the users.

2.1 Shutdown prediction techniques

Interestingly, spinning down the disk is not always beneficial. Accelerating the platters requires more energy than keeping them spinning while the disk is idle. Therefore, the time during which the device is off has to be long enough to offset the extra energy needed for the shutdown and spin-up sequence. This time is commonly referred to as the *breakeven-time*, and is usually on the order of a few seconds. Eliminating wrong shutdowns that not only waste energy but also significantly delay user requests is critical to conserving energy and reducing interactive delays. Simple timeout-based mechanisms gained wide popularity but they waste energy while waiting for a timeout to expire. As a result, various selections and dynamic adjustments of the timeout value have been proposed [18, 10, 14, 16] to reduce the amount of energy consumed during the timeout period. Consequently, dynamic predictors that shut down the device much earlier than the timeout mechanisms have been proposed to address energy consumption of the timeout period [6, 17, 27]. Stochastic modeling techniques have also been applied to model the idle periods in applications and shut down the disk based on the resulting models [2, 5, 23, 26].

To improve accuracy, energy management can be delegated to programmers, since they have a better idea of what the application, and potentially the users, are doing at a given time [11, 15, 20, 29]. To reduce the burden of hint insertion on the programmer, automatic generation of application hints was proposed [13] to exploit the observation that I/O activity is caused by unique call sites within applications. Finally, operating systems can concurrently evaluate multiple predictors and select the best one for the current workload [28].

2.2 Reducing spin-up delays

The goal of shutdown mechanisms powering down the disk is to improve energy savings. However, every shutdown requires a corresponding spin-up to serve future requests. It is important to note that even correct shutdowns can expose spin-up delays to the application or the user as shown in Figure 1. There are two approaches for reducing the impact of spin-up delays. First, we can prefetch and cache the data either in main memory [19, 24, 7] or an alternate storage device such as flash memory [21, 25], however disk accesses for uncached data will inevitably occur. Second, we propose waking up the disk early by spinning up the platters before the

request arrives and serving the request without any delays. Both approaches are complementary since the disk will have to be spun-up at some point even if the caching techniques are very efficient.

2.3 Predicting spin-up time

In this paper, we focus on spin-up prediction, which can be achieved in two ways. First, we can predict the length of the idle period and spin-up before the end of the predicted period. Second, we can predict spin-up itself by observing system events, such as user interactions. Prediction of the idle period lengths was previously proposed by Adaptive Learning Tree (ALT) [6]. The ALT approach is to predict the best current power mode based on a sequence of idle periods. Idle periods are discretized according to the time spent idling, and in relation to the number of available sleep states and device specifications. Previously observed states or sequences of states are encoded in a tree, the paths of which are matched according to newly observed sequences of discretized idle periods. Each leaf node in the tree constitutes a prediction and the most likely prediction is selected to transition the disk to the matching power state. ALT has shown significant improvement for power mode prediction in static, non-interactive applications and motivated us to adapt the design to predict the length of idle times and spin-up the disk before the predicted idle time ends.

ALT's discretization of idle periods depends on the number of available power states of the disk, and the prediction of the period lengths allows transition into shallower sleep states, where the disk's RPMs are reduced, but not halted, and the time to ready the disk is lessened. These periods are on the order of seconds and correspond to the breakeven time of each state. We can extend the design of ALT to predict longer idle periods and for the purpose of differentiation we will refer to our modified design as ALT+. The discretization of idle periods in ALT+ results not in the prediction for the best power mode, but rather for the duration of the current idle period. In order to spin up the disk in ALT+, we consider thresholds to be multiples of the disk's breakeven time, where the multiple is given by the discretization and encoded in the same tree structure as found in ALT. With this modification, ALT+ generates a likely time to ready the disk in anticipation of upcoming I/O activity, allowing the disk to be spun up on time to service the request.

2.4 Challenges in predicting spin-up time

Accurately spinning up the disk is challenging since the idle period has to be predicted very accurately. There are three possible situations that can occur following the prediction. The first and best scenario is when the pre-

diction is accurate and the disk is powered up just before the request arrives. In this situation, there is no energy wasted in waiting for the request to arrive and also the spin-up delay is hidden from the user. The second scenario occurs when the predicted idle period is longer than actual idle period. In this case, the device is powered-up upon I/O arrival and the latency of the spin-up is exposed to the user. The last scenario occurs when the predicted idle period is much shorter than the actual idle period. In this case, the disk is powered up and subsequently shut down without serving any disk requests. The disk is shut down to prevent it from remaining in the powered-up state for long idle periods. As a result, energy is wasted performing the unnecessary spin-up and shutdown transitions and spin-up delay is exposed when I/O requests do arrive.

Predicting the exact length of idle periods in interactive applications is difficult since it depends on the constantly changing frequency of user interactions with the application. Therefore, we propose to observe user interactions and infer from them the impending arrival of I/O activity, since users are responsible for the majority of I/O activity in interactive applications. Our mechanisms reconstruct the user's interaction context from mouse events directed at the application's GUI, thereby providing the necessary hints transparently and without application modification. The captured user context results in high accuracy and prediction timeliness in the proposed IASP.

3 Design

The key idea in IASP design is that the correlation between user interactions and disk I/O activity can be transparently exploited to predict I/O activity ahead of time and perform a timely disk spin-up to serve the request. Subsequently, our design faces several requirements:

- User interactions have to be captured transparently without modification of applications.
- Capture and prediction should be efficient to prevent excessive energy consumption by the CPU to train and generate predictions.
- The system should handle multiple applications in a graphically rich environment.
- User behavior correlation and classifications should be performed online and without direct user involvement.

The first three items are addressed by a novel implementation we propose in this paper. The last item is addressed by the proposed predictor design.

3.1 The Naïve Predictor

The observation that user interactions are responsible for the majority of disk I/O, in the interactive applications, leads us to a proposal of a simple mechanism that spins up the disk upon mouse clicks. The intuition dictates that if the user actively interacts with an application, which may require disk I/O, the disk should stay on to satisfy user requests. If the user is not actively interacting with the application the likelihood that the disk will be needed drops and the disk can be shut down. Therefore, our naïve All-Click Spin-Up mechanism (ACSU) spins the disk up upon each mouse click and keeps it spinning as long as the user is interacting with the application. Once the user stops interacting, the disk shuts down after a timeout period.

ACSU mechanisms act on all mouse clicks and spin up the disk as soon as possible, with the downside of unnecessary spin-ups for clicks that are not followed by any disk I/O. It is important to note that user interactions that require disk I/O are a small subset of all user interactions with the application. Therefore, ACSU mechanisms have the greatest potential for reducing spin-up delays at the expense of energy consumption caused by unnecessarily spinning the disk up and keeping the disk spun up without serving any disk I/Os. ACSU mechanisms set a lower bound on the spin-up delays for the proposed IASP and also illustrate the need for more intelligent prediction schemes that decide when the disk should be spun up to improve energy efficiency.

3.2 Capturing user interactions

The basis for the energy efficient design of IASP is the accurate and detailed monitoring of user activity. Most interactive applications are driven by simple point-and-click interactions. All operating systems targeted for consumers offer a Graphical User Interface (GUI) to facilitate uniform interfaces for interactions between users and application. As a result, virtually all interactions can be accomplished through mouse clicks [9]. Users interact with an application to accomplish specific tasks, like opening or saving a file. Many tasks can be accomplished by a single point and click, but other tasks require sequences of interactions. For example, to save a new file, the user commonly clicks on the File menu then selects the Save option and is presented by a directory selection menu, once the filename is entered and the user clicks OK, the file is saved and disk I/O may be requested. We argue that all GUI interactions resulting in disk I/O activity can be accurately captured and correlated to the activity they initiate, however, it should be noted that this does not always account for all occurring I/O activity.

User activity can be captured at various levels of de-

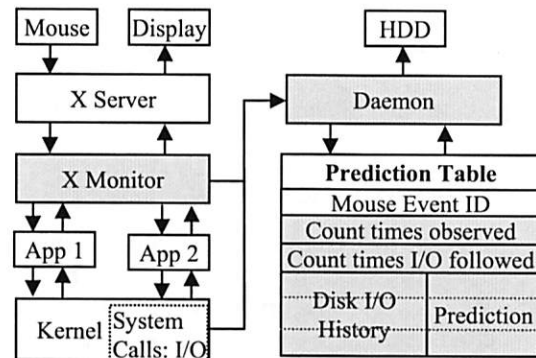


Figure 2: IASP architecture.

tail. The simplest method is to capture the coordinates of mouse clicks relative to the application window and approximate the graphical interface features from clusters of clicks [1]. However, this method either requires off-line processing or complicated on-line mechanisms for clustering of incoming clicks. Clustering is necessary since the only available information about user interactions are the relative coordinates of mouse events. K-means clustering is an effective approach, but suffers from several inefficiencies. First, the number of clusters has to be known a priori, meaning that for each application, we are faced with pre-determining the number of interactive elements. Second, assigning newly observed clicks to their respective clusters has a high processing overhead. Finally, each time the layout of the window changes, the mechanisms will generate mispredictions and also require retraining. This method is clearly far from the goals we set above, since it is neither very transparent nor computationally efficient. In order to address these problems, we turn to monitoring the GUI protocol streams and directly identifying the interactive elements.

On Unix-like systems, the X Window System is the common display protocol built on the client-server model. It is responsible for accepting graphical output requests from and reporting user input to clients. The stream of data from the client to the server contains the information about the window layout, while the data sent from the server to the client applications contains the information about user interactions. By adding an intermediary layer, as shown in Figure 2, between the server and its clients, we can observe the exact sequence of requests and events. This layer allows for transparent monitoring of user behavior. No modification of applications is necessary. Furthermore, user interactions are captured exactly, eliminating both the excessive computational overhead of computing a clustering and the inaccuracies associated with the clustering present in the previously described solution [1]. Since the need for cluster formation and behavior detection is eliminated, the offline process-

ing needed in the clustering approach is eliminated, fully allowing for detection, correlation, and prediction to be performed online.

The X Window System tracks GUI windows in trees, whose structure remains the same across executions of an application. Mouse event IDs are generated by storing and traversing the trees, generating a single unique integer for each node. The IDs generated by tree traversals are augmented with the size and location of the visual element that the mouse event occurs in. Information regarding the application's window tree structure is obtained from the X Window server. Internally, the server tracks window nodes and their children with identifiers that are unique to the application during an execution. Upon subsequent executions these server-side identifiers may change, but the structure of the tree remains the same. By performing a tree traversal and labeling all visited nodes in turn, we generate IDs based on a structural representation of the tree. This allows us to reuse the training across multiple executions of the application.

3.3 Monitoring & correlating I/O activity

Our mechanisms monitor each application individually for mouse clicks and file I/O. This allows a more accurate correlation of file I/O activity to user interactions with an application. We use two levels of correlations. First, the application's file I/O activity is captured by the kernel in the modified I/O system call functions that check for file I/Os. For example, we modified *sys_read* to check if the I/O call that entered *sys_read* is indeed file I/O since *sys_read* can be used for many types of I/O. This stage does not consider buffer cache effects since file I/O activity is captured before the buffer cache. As a result, we obtain a more accurate correlation between file I/O and mouse interactions. Second, once potential file I/O activity is detected, we follow the call to see if it resulted in an actual disk I/O or it was satisfied by the buffer cache. We use this information to correlate the user interactions that invoke file I/O to the actual disk I/O. We argue that the usage patterns in the buffer cache will also correlate to the user interactions, since user behavior is repetitive, and we show that IASP is able to predict actual disk I/Os with a high degree of accuracy.

3.3.1 Correlating file I/O activity

We record correlation statistics in the prediction table that is organized as a hash table indexed by the hash calculated using the mouse event IDs. Figure 2 shows the prediction table organization and the content of the table entry that is maintained by the IASP daemon. The click IDs are unique to the window organization and therefore do not result in aliasing between different applications

Observed Clicks	Click ID	Bookkeeping	Actions
File	C1	C1	Sequence of clicks is being recorded
Page Setup	C2	C1,C2	
Portrait	C3	C1,C2,C3	
OK	C4	C1,C2,C3,C4	
File	C1	C1	Longer idle period, potentially signaling end of sequence
Open	C5	C1,C5	
File Select	C6	C1,C5,C6	C1 – root of the tree observed, sequence collection restarted
Open	C7	C1,C5,C6,C7	

DISK I/O

Figure 3: Example of interaction sequences.

and windows as explained earlier. The data stored by the prediction table contains only the unique event ID, the number of times the event was observed, and the number of times I/O activity followed. The counts are a simple, but efficient means of computing an empirical probability for future predictions. The table resides globally in a daemon and is shared among processes to allow table reuse across multiple or concurrent executions of the application. Furthermore, the table can be easily retained in the kernel across multiple executions of the application due to its small size.

In addition to the global prediction table, IASP records the history of recent click activity for each process in the system. Consider a typical usage scenario shown in Figure 3 where a user is editing a file in a word processor. After a while, the user clicks through a file menu to change properties of the edited file. The recorded history of clicks is C1, C2, C3, C4. At this point, the user decides to work on the file again. If the time is long enough we can consider the clicks to be uncorrelated and the history of clicks is cleared. Alternatively, the user may immediately proceed to open a new file with click sequence of C1, C5, C6, C7. In this case, the history is also reset when the user clicks on C1. Since all menus are organized as trees in the application, clicking on C1 signifies return to the root of the File menu tree. Therefore, when IASP detect a repeated click ID in the history, the history is restarted with the current click. It is still possible to record uncorrelated clicks in the history. For example, user interacts with the Edit menu and subsequently opens a file. In this case, the history will contain clicks for edit menu interactions and the file open interactions. However, the uncorrelated clicks will have low probability and will eventually be made insignificant to the predictor with further training.

IASP uses very simple training where the observed count is updated every time a particular click is detected. In order to correlate file I/O activity, the history of clicks that lead to file I/O is traversed and the I/O count for ev-

ery click present in the history is incremented. Ratio of both counts gives us the probability of file I/O following the particular click.

3.3.2 Correlating disk I/O activity

File I/Os issued by the application can be satisfied by the buffer cache and as a result may not require any disk I/O and the disk can remain in a power saving state. Since not all file I/Os result in disk I/O, we introduce an additional correlation step to correlate the mouse click to the disk I/O. We use a history of file I/Os generated by the particular click IDs to predict the future disk I/O generated by the particular click. We add a 2-bit history table with a 2-bit saturating counter to record the history of file I/Os that resulted in disk I/O after a given mouse click was observed. We update the prediction table using the history of file I/Os and the resulting disk I/O and the current outcome of the file I/O. Combination of the file I/O probability and the resulting I/O prediction results in a final decision about disk spin-up.

We are relying on file I/O prediction and disk I/O prediction to separate application behavior from the file cache behavior. By considering both the probability of a particular click being followed by file I/O and the behavior of the resulting I/O in the buffer cache, IASP can accurately discern whether that click will result in actual disk I/O. Separating the predictor's training into file I/O and buffer cache behavior allows accurate correlation of clicks to the application's file I/O, which is the fundamental goal of this paper. Mouse interactions with the application's GUI are strongly correlated to file I/O, so, intuitively, the goal of the described implementation is to filter all uncorrelated clicks first before the buffer cache impact on disk I/O is considered. Finally, we use a simple 2-bit history to predict buffer cache behavior, which provided sufficient accuracy. However, more sophisticated buffer-cache behavior prediction can be potentially employed to further improve IASP accuracies.

3.4 To predict or not to predict

The critical issues that we are addressing in our design are timeliness and accuracy, which turn out to be competing optimizations. Many application functions can be invoked with just a single click, however certain operation may require several steps. In case of multiple clicks, the last click initiates a system action that is a response to the user's interaction. More specifically, we can observe only the last click just before disk I/O occurred and correlate the click to the particular disk behavior with high accuracy. While this approach is very accurate, it is not very timely. Correlating disk I/O to the last click occurring before the I/O request was observed does not

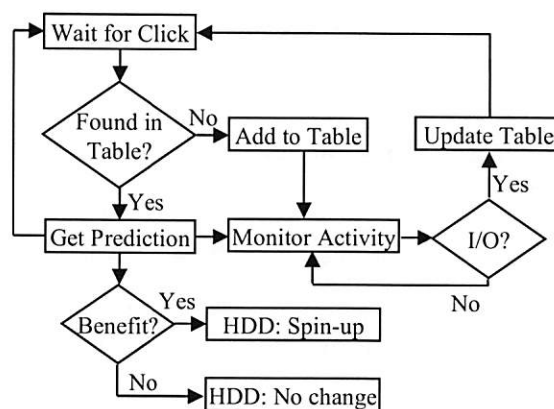


Figure 4: IASP decision flowchart.

provide adequate time before the I/O arrives to offset a significant portion of the spin-up latency, and so has a negligible impact on reducing the associated interactive delays. This scenario is illustrated in Figure 3. Clicking on C7, which is the final Open button in file open sequence, will result in an I/O system call. However, the click is immediately followed by I/O and waiting for prediction until last click will provide little benefit in reducing delays exposed to users. Spinning up the disk upon C1 click provides sufficient time to reduce delays; however, it may result in erroneous spin-ups, since the user may perform other operations that do not lead to file I/O.

3.5 Predicting upcoming activity

Figure 4 shows the decision-making process. Upon each mouse click, we use the ID of the current click to calculate the prediction table hash index. The daemon performs a prediction table lookup with three possible outcomes: the entry is found indicating that the interaction leads to file I/O with probability above the threshold, the corresponding entry is found but contains a low probability of upcoming file I/O, or the entry is not found. Events which are not found in the table are added and updated accordingly to our training routine described earlier. Once the entry is found that satisfies the desired probability threshold, we use the current history of disk I/Os for the given mouse click and perform final lookup into history prediction table for the selected click. The prediction from the history table dictates the outcome of the disk state and the disk is transition to the predicted state.

In our experimental implementation, we consider only two possible states for the disk, standby and sleep. Therefore, the decision to spin-up the disk is binary, i.e. the determination is made that either I/O activity is forthcoming following the mouse event or it is not. However, our binary decision predictor can easily be ex-

Appl.	Number of I/O Periods	Read (MB) Without Cache	Write (MB) Without Cache	Read (MB) With Cache	Write (MB) With Cache	Number of Clicks	Number of IDs
<i>Firefox</i>	814	1903.35	350.8	851.91	120.39	3857	130
<i>Writer</i>	1385	2043.62	2186.28	1434.05	2120.47	5755	195
<i>Impress</i>	1485	1230.42	263.6	517.06	60.44	25375	194
<i>Calc</i>	2846	1840.4	116.7	1280.7	59.67	9102	35
<i>Gimp</i>	844	1443.32	957.3	796.9	936.54	8465	157
<i>Dia</i>	6362	174.31	65.3	123.64	10.28	46864	118

Table 1: Applications and execution details

tended to devices with multiple sleep states. The only required modification is the discretization of the probabilities computed from the prediction table. Consider the case of 2 sleep states, one having the platters fully spun-down, and the other having the platters spinning at a reduced RPM. The third possible state is full idle. This scenario is easily handled by adding a second threshold. With two thresholds, probability values falling beneath the lower one generate a prediction favoring the halted platters sleep state. Probability values falling between the two thresholds would cause the disk to enter the reduced RPM sleep state. Finally, any probability values above the higher threshold would fully spin-up the disk. Clearly this modification can be extended to an arbitrary number of sleep states.

3.6 Multiprogramming environment

As described, our mechanisms work in a multiprocess environment since training and prediction are made independently of other processes. The described monitoring mechanism allows us to uniquely identify windows from multiple processes and allow for accurate correlation without any aliasing from other applications. The prediction performed by IASP is also easily integrated into a multiprocess environment since as soon as IASP predict spin-up for a single process the disk is spun up without considering other processes. This is opposite of the shutdown mechanisms which have to consider other processes that are currently running and may need the disk. In case of spin-up, once the disk is needed it has to be spun-up.

4 Methodology

We evaluate the performance of ACSU and IASP mechanisms, comparing them to ALT+. In order to fully evaluate the effectiveness of our proposed mechanisms, we use a trace-based simulator as well as an implementation of the mechanisms that replays the traces in real time with an actual disk. We focus on predicting spin-ups

State	WD2500JD	40GNX
Read/Write Power	10.6W	2.5W
Seek Power	13.25W	2.6W
Idle Power	10W	1.3W
Standby Power	1.8W	0.25W
Spin-up Energy	148.5J	17.1J
Shutdown Energy	6.4J	1.08J
State Transition		
Spin-up time	9 sec.	4.5 sec.
Shutdown time	4 sec.	0.35 sec

Table 2: Disk energy consumption specifications.

and as a result, we use a simple timeout based shutdown mechanism with the timeout set to 20 seconds which is comparable to the breakeven time of both disks. This means that the disk is shut down after 20 seconds of idleness. This also applies to erroneous spin-ups, where when the disk is spun up, it waits for the timeout to expire before subsequently shutting down.

Detailed traces of user-interactive sessions for each application were obtained by a modified `strace` utility over a number of days. The modified `strace` utility allows us to obtain the PID, access type, time, file descriptor, as well as the amount of data that is fetched for each I/O operation. The specifications of the simulated disks belong to Western Digital Caviar WD2500JD and Hitachi Travelstar 40GNX hard drives and are shown in Table 2. The WD2500JD has a spin-up time of about 9 seconds from the sleep state, the surprising duration of which appears to be remarkably common for high-speed commodity drives. The 40GNX is designed for portable systems and as such has much lower energy consumption and spin-up time than the WD2500JD.

Table 1 shows six popular desktop applications chosen for our evaluation: *Firefox*, *Writer*, *Impress*, *Calc*, *Gimp*, and *Dia*. *Firefox* is a web browser with which a user spends time reading page content and following links. In this case, I/O behavior depends on the content of the page and user behavior. *Impress* (presentation editor), *Writer* (word processor), and *Calc* (spreadsheet editor),

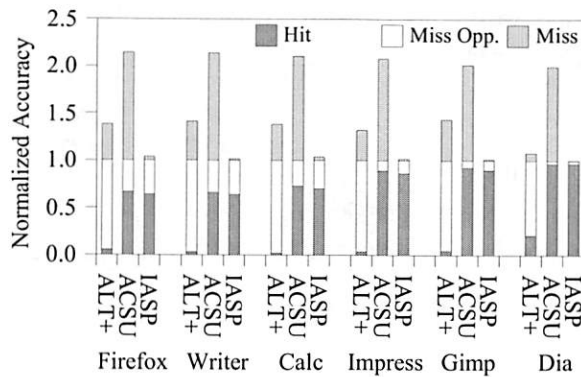


Figure 5: Prediction accuracy normalized to the total number of disk spin-ups *without* the buffer cache.

are part of the Open Office suite of applications. All three are interactive applications with both user driven I/O and periodic automated I/O, i.e. autosaves. *Gimp* is an image manipulation program used to prepare and edit figures, graphs, and photos. Finally, *Dia* is an application used for drawing diagrams for papers and presentations.

Table 1 also lists the total number of idle periods for which a potential shutdown and a corresponding spin-up are required, the total amount of read and write activity, a total number of mouse click interactions and the number of unique click interactions encountered in the studied applications. We show statistics for I/O requests that are generated by the application (shown as 'Without Cache') and I/O requests that are not filtered by the buffer cache and are sent to the disk (shown as 'With Cache'). In our experiments, we use an LRU managed buffer cache of size 512MB, which is representative of current systems' capabilities.

5 Results

5.1 File I/O Correlation Accuracy

Accurate prediction ensures that the disk is not spun-up needlessly, when no activity is forthcoming. We first consider the accuracy of correlating mouse clicks to file I/O at the application level, before it is filtered by the buffer cache. Figure 5 shows the breakdown of correct and incorrect spin-ups, i.e. hits and misses, for ACSU, IASP and ALT+ that result from predicting file I/O when the system does not employ buffer caching. Hits are counted when the prediction to spin-up the disk is made and it is followed by file I/O. Misses are those spin-ups which were not followed by any I/O, and Missed Opportunities are periods for which the mechanism failed to provide a prediction, but a spin-up was needed. Each missed opportunity results in the disk being spun up on

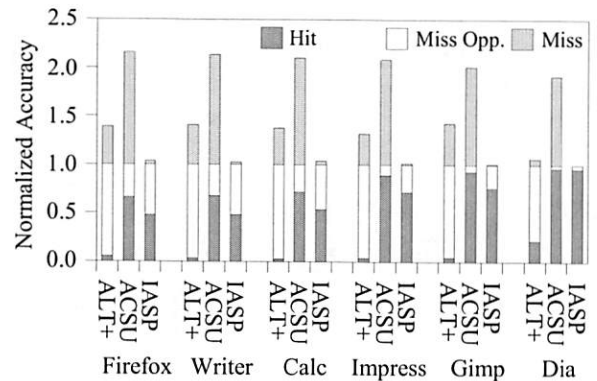


Figure 6: Prediction accuracy normalized to the total number of spin-ups *with* the buffer cache.

demand, essentially spinning up when an I/O request arrives. ACSU mechanisms keep the disk powered up while a user is interacting with the application, minimizing the interactive delays. While it provides an upper bound for the number of I/O periods that may be predicted by clicks (i.e. the number of periods covered by IASP can equal but not exceed the number of periods covered by ACSU), it naïvely spins-up the disk for all clicks, resulting in excess misses. Low coverage and high inaccuracies in ALT+ illustrate the behavior of the mechanisms that solely rely on observing system events without considering user interactions.

ACSU on average covers 81% of all file I/O periods, while IASP correctly covers an average of 79% of periods. The lack of contextual information and the random nature of idle period duration results in ALT+ correctly covering an average of 7% of periods. ACSU shows the greatest number of misses for all applications, 52% of spin-ups are misses. This miss rate reflects the number of existing mouse clicks that do not correlate to any I/O. When the disk is spun-up in ACSU, it will remain spinning as long as new clicks are observed and the idle threshold is not reached between any two clicks. IASP consistently results in the fewest misses, averaging 2%, which mostly occur while the predictor is warming up.

Whereas ALT+ considers solely I/O patterns when generating predictions, coverage by the ACSU and IASP mechanisms is contingent on the availability of mouse events preceding I/O activity. *Firefox*, *Writer*, and *Calc* show the greatest number of misses and missed opportunities for both ACSU and IASP, meaning that there is a good deal more ambiguity in the mouse events available for prediction generated by these applications than the others. In the case of *Firefox*, most mouse activity occurs within the window displaying the visited web pages. As such, the constantly changing structure of the window increases the number of mouse IDs that are encountered

resulting in a high misprediction rate for ACSU. IASP, on the other hand, does not spin-up the disk for these clicks, since their IDs are not observed as often as those that belong to the static part of the GUI. In the case of *Writer* and *Calc*, the relatively low coverage by both ACSU and IASP mechanisms is caused by lower availability of clicks preceding I/O. While most or all functionality of these applications is accessible through the GUI, the interaction is made simpler through the use of keyboard shortcuts. As we are only considering mouse events, any I/O that occurs in response to a keyboard shortcut is not predicted by the mechanisms. The applicability of keyboard events to I/O prediction will be explored in future research.

The applications for which both ACSU and IASP perform best are *Impress*, *Dia* and *Gimp*. These applications have more complex GUIs for which extensive keyboard shortcuts are not intuitive to an average user. This is the case with *Dia* and especially *Gimp*. All three of these applications also depend heavily upon the mouse, due to the graphical nature of their content and usage. Manipulating images, graphs, and figures is done most easily with the mouse and in our traces the user depended more heavily on the mouse for all interactions with these applications.

5.2 Impact of the Buffer Cache

High file I/O prediction accuracy shown in Figure 5 represents the strong correlation between mouse clicks and file I/O. However, file I/O may also be satisfied by a buffer cache access, making a disk spin-up unnecessary. Hence, we have to consider the impact of the buffer cache on prediction accuracy. From this point on, all figures show the mechanism with the buffer cache enabled. We set the buffer cache to 512MB, which is representative of current systems' capabilities. The buffer cache can satisfy many file I/Os resulting in fewer required disk accesses. In addition to introducing additional randomness into file I/O patterns, the buffer cache also increases the training time of prediction mechanisms due to both inclusion of the access history and fewer spin-ups encountered in the system. Similarly to Figure 5, Figure 6 shows hits, misses, and missed opportunities, but those metrics now realistically reflect the actual required disk I/O.

The aggressiveness of our ACSU mechanism again makes it a top performer when we consider the amount of periods covered with correct spin-ups. This behavior can be expected since the mechanism just keeps the disk on no matter if the buffer cache satisfies the request or not. Therefore, this mechanism's behavior is not impacted by the presence of the buffer cache. The different fraction of period misses, and hits, as compared to Fig-

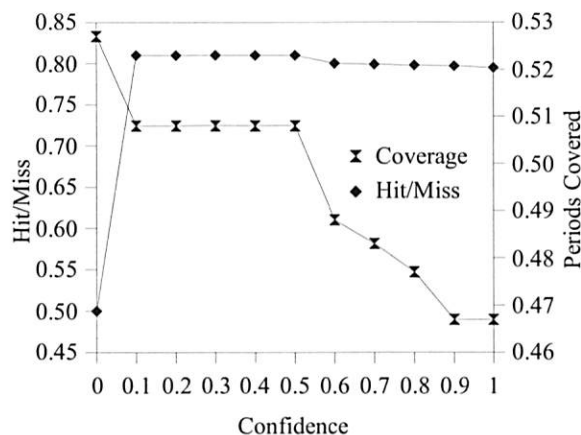


Figure 7: Hit/Miss ratio and I/O activity period coverage vary as the acceptable confidence level is increased.

ure 5 are due to a change in the periods' composition since we have fewer and longer periods due to filtering of I/O by the buffer cache. In this case, ACSU is able to spin up the disk ahead of time for 66% of required periods, while incurring misprediction rates as high as 54% with the average of 52%. The ALT+ mechanism is also not impacted much by the buffer cache since the randomness observed by the file I/O in the interactive application is already large rendering this mechanism not very useful in either case. The coverage is as low as 3% with an average of 7%, incurring a high misprediction rate of 24% on average.

The impact is more pronounced in the case of IASP, since IASP uses contextual prediction selectively to predict what user activity will result in disk I/O. Therefore, the introduction of any randomness by the buffer cache affects the accuracy of our history-based IASP disk I/O prediction. IASP remains the most accurate mechanism, resulting in only 2% mispredictions while achieving 65% of correct spin-ups, on average. Low misprediction rate indicates that the randomness introduced by the buffer cache is insignificant and the history-based prediction is able to capture correctly the behavior of the disk I/O. Lower coverage indicates that the fewer I/O periods increase the fraction of learning time. It is worth noting that the significance of learning time decreases the longer the system stays on.

In the case of all applications except for *Dia*, the lower coverage of the IASP mechanism as compared to the coverage of the uncached I/O is due to learning, since predictions followed by an absence of disk I/O due to caching result in fewer learning opportunities. Interaction with GUI elements results in the requisite file data being stored in the cache. In the absence of a cache, even the infrequently used elements would generate disk

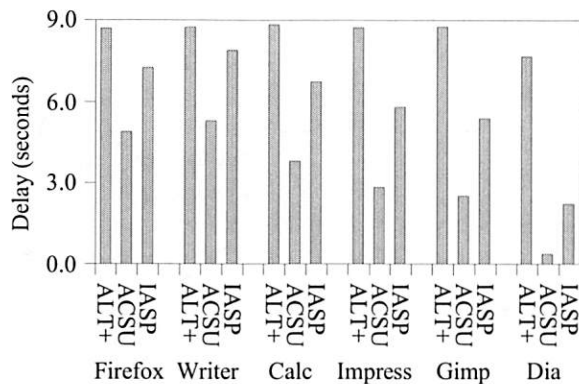


Figure 8: Average delay in seconds, WD2500JD.

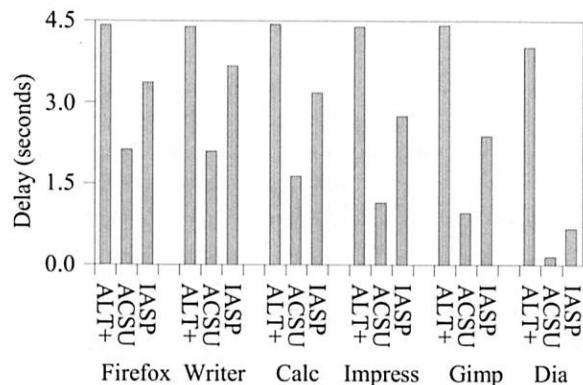


Figure 9: Average delay in seconds, 40GNX.

I/O, but not so with the cache. In general, IASP greatly reduces the number of unnecessary spin-ups that are present in ACSU, at the cost of lower coverage, due to more energy-efficient spin-up policies.

In the case of *Dia*, the type of interactions encountered during tracing were limited to very simple actions, such as opening, creating and saving a number of files containing various simple figures, meaning that the availability or absence of I/O was quickly learned by IASP. Creating even the simplest diagrams may require a large number of clicks. ACSU therefore exhibits a large number of mispredictions in this case, while IASP easily filters out the events that cause the program to, for example, draw a triangle rather than open a file.

5.3 Confidence Levels

Confidence levels are dynamically set thresholds for prediction within IASP. Recall that confidence levels associated with the mouse events represent the ratio of how many times the event was observed and the number of times the event was followed by file I/O. A given confidence level dictates the amount of predictions made and the prediction accuracy as illustrated in Figure 7. Confidence of 1 means that the click is always followed by I/O activity, confidence of .9 means that the click is followed by I/O activity 90% of the time, and so on. If the confidence level is set too low, the predictor may spin-up the disk early in response to events that rarely lead to I/O activity. For example, the user clicked on File menu but interacted with options that did not involve disk I/O. However, early spin-ups hide more latency if the interaction leads to disk I/O. Setting the confidence level too high, however, may delay the disk spin-up and potentially expose the entire spin-up latency to the users. It is therefore important to set confidence levels such that the energy consumption caused by early-erroneous spin-ups and the delay reduction offered by the early spin-ups are

in balance.

Figure 7 illustrates impact of confidence on Hit/Miss ratios and file I/O period coverages. We define coverage as a fraction of correctly predicted spin-ups in the applications. The ratio of hits to misses shows the average accuracy over all applications in predicting upcoming I/O activity. We see that due to optimistic prediction, the hit/miss ratio declines slightly as the acceptable confidence level increases past 0.5, but overall remains steady at just over 80%. The sharp increase and steady behavior in the hit/miss ratio indicates quick convergence during training and stable behavior for each mouse event. Increasing confidence level past 0.5 results in longer training and the predictor's coverage drops sharply, since it attempts to predict fewer and fewer disk spin-ups.

5.4 Delay Reduction

Figures 8 and 9 show the average spin-up delays that are exposed to the user in the case of each of the two disks. Each missed opportunity seen in Figure 6 results in delay equal to the average spin-up time, 9 seconds in the case of WD2500JD and 4.5 seconds in the case of 40GNX. On the other hand, hits described in Figure 6 are predictions that result in the disk spinning up correctly and may arrive either early enough to allow the disk to spin-up before the I/O arrives, resulting in no delay, or late, where the disk is in the process of spinning up when I/O arrives. The demand based spin-up exposes full spin-up delay to the application, during every spin-up, and therefore the average delay in demand based system is full 9 seconds in the case of WD2500JD and 4.5 seconds in the case of 40GNX.

ACSU is very aggressive in reducing spin-up delays at the expense of increased energy consumption. High coverage of file I/O periods in Figure 6 results in an average spin-up delay reduction from 9 seconds to 3.3, which is only 37% of the spin-up delay exposed by the

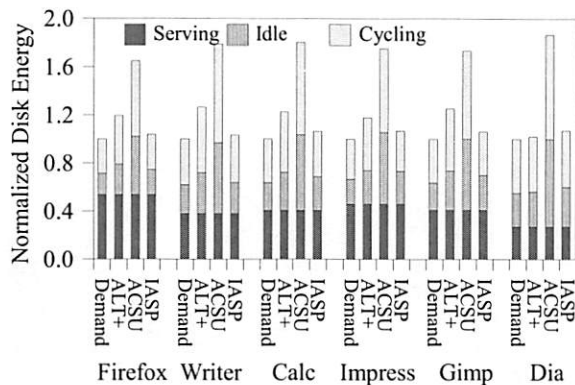


Figure 10: Energy consumption, WD2500JD.

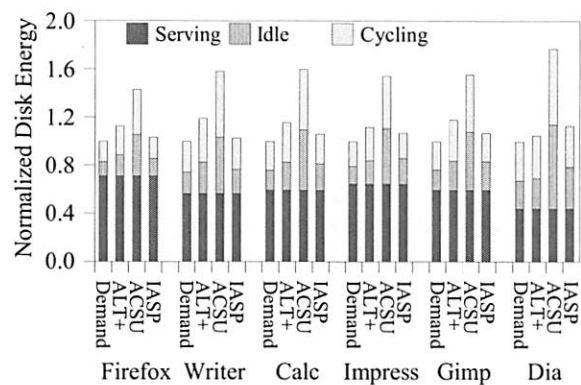


Figure 11: Energy consumption, 40GNX.

demand-based spin-up for the WD2500JD. In the case of the 40GNX disk, shown in Figure 9, the average spin-up delay is reduced to 1.36 seconds, which is 30% of demand-based spin-up delay. As expected from Figure 6 ALT+ performs poorly exposing high delays of 8.58 seconds and 4.35 seconds for the WD2500JD and 40GNX, respectively. The exposed delays in ALT+ are comparable to demand based spin-up delays, since most periods require on-demand spin-up.

IASP is able to shorten interactive delays exposed to the users down to 5.89 seconds and 2.67 seconds for WD2500JD and 40GNX, respectively, while maintaining high accuracy and low energy consumption. IASP exposes 2.6 seconds and 1.3 seconds more delay than ACSU for WD2500JD and 40GNX, respectively. ACSU sets the lower bound on the spin-up delay for mechanisms that utilize mouse interaction since it spins up or keeps the disk on for all mouse interaction as shown by the higher coverage in Figure 6. ACSU not only captures more I/O periods, but also does so earlier than IASP, since it is not governed by the confidence requirement set in IASP to prevent erroneous spin-ups. ACSU is therefore most effective in situations where low delay is desired, assuming that of course energy-efficiency is also a desired attribute, but to a lesser extent. The higher accuracy of IASP makes it the most desirable choice when energy efficiency is important and users are willing to tolerate slightly higher delays than ACSU provides, which are still much lower than delays exposed by the demand-based spin-ups.

Highest delay reduction is present in *Dia*, where the delay is reduced by 93% and 85% by ACSU and IASP for 40GNX, and 96% and 85% for the WD2500JD, indicating that there is plenty of user think time to overlap spin-up delays. On the other hand, *Writer* shows the lowest reduction in spin-up delays. The most significant factor contributing to the low reduction in spin-up delays in case of *Writer* is single button interaction with tool-

bars, which results in I/O activity. For example, if the user clicks on the spell-check button in the toolbar rather than finding spell-check in the Tools menu, the resulting activity arrives quickly following the single mouse event that predicted it.

Reduction in delay is generally accompanied by increase in energy consumption, since we need the disk to remain on in order to minimize the delay. For example, if we allow the disk to remain spinning for the entirety of an application's run, the interactive delays are eliminated, but at the cost of vastly increased disk energy consumption. On the other hand, simple demand-based mechanisms are often the lowest energy solution, due to the fact that they do not have extraneous spin-ups, but they incur delays each time the disk is spun up. We have seen that delay can be significantly improved using our proposed ACSU and IASP mechanisms and now we turn to a discussion of energy consumption.

5.5 Energy

Figures 10 and 11 show the details of energy consumption of the two disks. The energy consumption is divided into I/O serving energy, power-cycle energy, and idle energy. I/O serving energy is consumed by the disk while reading, writing, and seeking data. I/O serving energy is the same for all mechanisms, since the amount of I/O served is the same. Power-cycle energy is consumed by the disk during spin-up and shutdown and is directly related to number of spin-ups which also include erroneous spin-ups. Finally, idle energy is the energy consumed by the disk while it is spinning but not serving any I/Os. Idle energy is dependent on the number of I/O periods and the timeout before the disk is shutdown after an I/O period, additional idle energy consumption occurs in ACSU, IASP, and ALT+ due to mispredictions, during which the disk idles before shutting down when I/O does not arrive. In addition, early spin-ups result in ad-

ditional energy being consumed by the disk between the time when the disk is ready to serve data and the arrival of the first I/O, which is most prevalent in ACSU.

Due to a large number of mispredictions, ACSU consumes significantly more idle and power-cycle energy than IASP. On average, IASP consumes 30% less energy idling than ACSU, and 40% less energy cycling power modes when using WD2500JD. In 40GNX's case, IASP consumes 27% less idle energy than ACSU, and 25% less cycling energy. On average, with the WD2500JD, IASP consumes 6% more energy than the on-demand mechanism due to waiting after early spin-ups and the few mispredictions that result in the consumption of energy not present in the on-demand mechanism. Similarly, in the 40GNX case, IASP consumes 7% more energy than the on-demand mechanism. Keeping the disk always on has the effect of increasing idle energy consumption to levels that are prohibitively large for energy constrained systems. Overall, the energy consumed by WD2500JD using ACSU is 49% lower than keeping the disk always on, 70% lower in case of IASP, and 65% lower for ALT+. The energy consumed by 40GNX when using IASP is 64% lower, 60% lower for ALT+. Differences in relative energy consumption result from the different power profiles of the two disks in question.

5.6 Overheads

The computational and storage overheads of any power management mechanism have to be taken into consideration, since improving the energy consumption of one device while equally increasing that of another does not result in energy-efficiency. Therefore, it is critical to keep computational requirements to minimum to avoid the excess energy consumption in the processor. Additionally, the storage overheads of a power management mechanism's data should be low enough to be considered insignificant, since storing a large amount of data could potentially impact the execution of interactive applications by polluting data caches in the processor.

Considering those requirements, ACSU has a clear advantage since it does not have to store or compute anything. It simply spins the disk up when the disk is shut-down and a click arrives or resets the timeout variable when the disk is active. IASP, on the other hand, computes IDs that uniquely identify user mouse interactions and store the interaction predictions in a prediction table. Due to the efficiency of hash tables, the only measurable computational overhead is incurred when the unique window ID is computed. *Firefox* has the deepest tree of 27 levels in the studied applications. Therefore, we setup an experiment to measure the average overhead of traversing 27 levels of tree hierarchy and we found the overhead to be negligible. Furthermore, this overhead

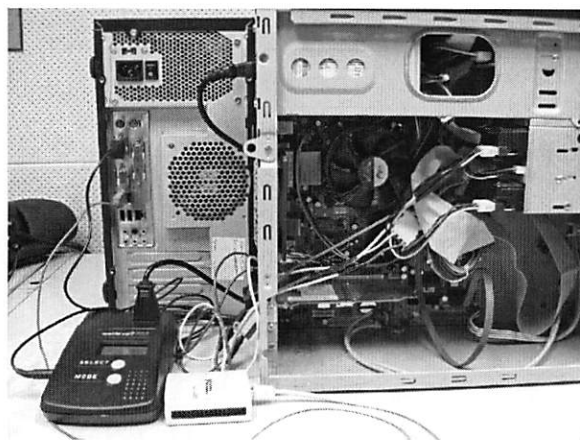


Figure 12: The experimental setup used for measuring power.

can be almost eliminated by modifying the X-Window Server to automatically generate mouse click IDs as it itself builds the window tree, rather than building a separate representation as shown in Figure 2.

The storage overhead is likewise relatively low in IASP. For each unique mouse event we are storing its ID (32 bits), the number of times the event was observed (32 bits), the number of times it was followed by I/O activity (32 bits), and the two-bit history table (8 bits) with two bit saturating counters for the prediction outcome (8 bits). The resulting table entry is 14 bytes. The number of unique click IDs in the studied applications ranged widely from 35 in *Calc* to 195 in *Writer*. Therefore, in the worst case *Writer* requires 2.67KB to store 195 entries. An 11.3KB table would suffice for storing all entries from every one of the six applications we have studied.

5.7 Experimental Evaluation

Experiments were conducted using a setup of two desktop machines with dual-core 3.0GHz processors and 2GB of memory. As shown in Figure 12, a multi-channel data acquisition board (DAQ) from NI was connected to the power cable of a WD2500JD hard drive dedicated to replaying the traces. To measure the power consumed, a 0.1 ohms resistor was placed in series with the hard disk power supply and the voltage drop across the resistor was fed to the DAQ. The second machine, running Windows XP and the DAQ drivers, ran the LabView setup sampling measurements at 1000Hz from the DAQ. The simulated trace-driven prediction mechanisms were ported to a driver that replays the traces on the measured hard drive.

Figure 13 shows a selected portion of the *Dia* trace, as replayed on the hardware and captured through the experimental setup. We show several activity periods for on-demand spin-up, ACSU, and IASP. We omit ALT+

due to its low accuracy. Figure 13 shows several Lines C, E, and G which show the spin-ups that were initiated on-demand when the I/O arrived. The period of disk activity beginning at A, as initiated by ACSU, illustrates ACSU's aggressive spin-up policy. At A the user begins a series of interactions which cause the disk to spin up and remain spinning until the interaction ends and I/O is served. The I/O arrived at C and the disk was spun up on-demand to serve the request. IASP, on the other hand, spun-up early at B, in response to a user interaction that predicted that I/O will follow.

Similarly, we see that the next interaction at D, caused both IASP and ACSU to spin-up the disk ahead of E, where the disk was spun up on-demand. Another example occurs at G, with on-demand spin-up shortly following the IASP and ACSU spin-ups. Matching up the trace replay to the simulator output, we have verified that this behavior is indeed expected and due to space constraints we include only limited results from the hardware replay.

6 Conclusion

In this paper, we proposed two disk spin-up mechanisms: ACSU that simply keeps the disk powered when users are interacting with the application and IASP that accurately and efficiently monitors user behavior. Both mechanisms reduce interactive delays exposed to the users due to energy management in hard disk drives. Hard disks contribute significantly to the overall energy consumption in computer systems. Therefore, aggressive energy management techniques attempt to maintain the hard drive in a low power state as much as possible exposing long latency spin-up delays to the users. Reducing the spin-up delays provides twofold benefit. First, the users are less irritated by constant lags in the responsiveness of the system due to disk spin-ups. Second, shorter delays allow the system to accomplish tasks quicker resulting in less energy being consumed by other components that have to wait for the disk spin-up.

The key observation used in our design is that users are responsible for the demand placed on the system through interactive applications. Therefore, monitoring user interaction patterns with applications provides the opportunity for predicting I/O requests that follow the interactions and use this to spin-up the disk ahead of time, reducing delays. Our evaluation of proposed ACSU and IASP shows significant improvement over modified ALT+ mechanisms in terms of predicting upcoming disk I/O activity and thereby shortening the interactive delay associated with energy management. ALT+ mechanisms are not able to accurately predict I/O activity in interactive applications resulting in an average misprediction rate of 25% that increases energy consumption in the system without providing any benefit of reducing

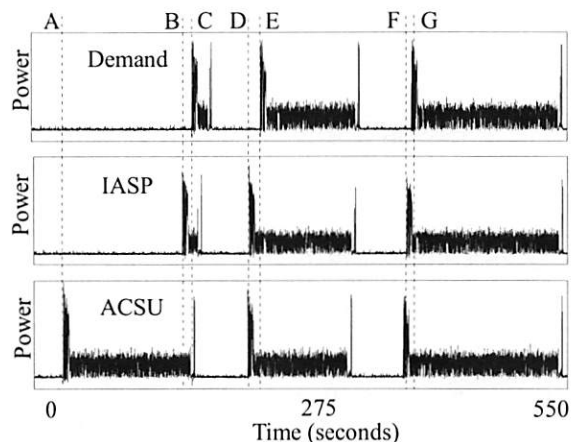


Figure 13: Power consumption in a selected 550 second period from *Dia* under Demand-based spin-up, ACSU, and IASP.

delays with only 7% of periods correctly predicted spin-ups, on average. ACSU mechanisms are very aggressive and achieve 81% of accurate predictions that reduce delays at a cost of 52% misprediction rate. As a result, ACSU is able to reduce spin-up delays on average by over 60% (over 5 seconds), albeit at the cost high energy consumption. Finally, IASP is much more accurate since it monitors user interactions. IASP on average achieves 79% of accurate predictions that reduce delays with only 2% of mispredictions. As a result, IASP is able to reduce spin-up delays on average by 35% (over 3 seconds), while maintaining low energy consumption.

The primary goal of this paper was to reduce interactive delays due to disk spin-up exposed to the users, while maintaining the energy efficiency of the shutdown mechanism. Spin-up mechanisms do not reduce energy consumption of the individual device, however they have a side effect of making the system more energy efficient by accomplishing tasks quicker and reducing the energy consumed by the system waiting for the disk to spin-up.

References

- [1] ALBINALI, F., AND GNIADY, C. CPM: Context-aware power management in wlns. In *Proceedings of the Eighteenth Innovative Applications of Artificial Intelligence Conference (IAAI)* (2006).
- [2] BENINI, L., BOGLIOLO, A., PALEOLOGO, G. A., AND MICHELI, G. D. Policy optimization for dynamic power management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18, 6 (June 1999), 813–833.
- [3] BIANCHINI, R., AND RAJAMONY, R. Power and energy management for server systems. Tech. Rep. DCS-TR-528, Department of Computer Science, Rutgers University, June 2003.
- [4] CHASE, J., ANDERSON, D., THACKAR, P., VAHDAT, A., AND BOYLE, R. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating Systems Principles* (October 2001).

- [5] CHUNG, E.-Y., BENINI, L., BOGLIOLO, A., LU, Y.-H., AND MICHELI, G. D. Dynamic power management for nonstationary service requests. *IEEE Transactions on Computers* 51, 11 (November 2002), 1345–1361.
- [6] CHUNG, E.-Y., BENINI, L., AND MICHELI, G. D. Dynamic power management using adaptive learning tree. In *Proceedings of the International Conference on Computer-Aided Design* (November 1999), pp. 274–279.
- [7] CRAVEN, M., AND AMER, A. Predictive reduction of power and latency (purple). In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 237–244.
- [8] CRK, I., BI, M., AND GNIADY, C. Interaction-aware energy management for wireless network cards. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2008).
- [9] DIX, A., FINLEY, J., ABOWD, G., AND BEALE, R. *Human-computer interaction (3rd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.
- [10] DOUGLIS, F., KRISHNAN, P., AND BERSHAD, B. Adaptive disk spin-down policies for mobile computers. In *Proceedings 2nd USENIX Symp. on Mobile and Location-Independent Computing* (1995), pp. 381–413.
- [11] ELLIS, C. S. The case for higher-level power management. In *Workshop on Hot Topics in Operating Systems* (Rio Rico, AZ, USA, March 1999), pp. 162–167.
- [12] ELNOZAHY, M., KISTLER, M., AND RAJAMONY, R. Energy conservation policies for web servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (March 2003).
- [13] GNIADY, C., HU, Y. C., AND LU, Y.-H. Program counter based techniques for dynamic power management. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (Dec. 2004).
- [14] GOLDING, R. A., II, P. B., STAELEN, C., SULLIVAN, T., AND WILKES, J. Idleness is not sloth. In *Proceedings of the USENIX Winter Conference* (1995), pp. 201–212.
- [15] HEATH, T., PINHEIRO, E., HOM, J., KREMER, U., AND BIANCHINI, R. Application transformations for energy and performance-aware device management. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques* (September 2002).
- [16] HELMBOLD, D. P., LONG, D. D. E., AND SHERROD, B. A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking* (1996), pp. 130–142.
- [17] HWANG, C.-H., AND WU, A. C. A predictive system shutdown method for energy saving of event driven computation. *ACM Transactions on Design Automation of Electronic Systems* 5, 2 (April 2000), 226–241.
- [18] KARLIN, A. R., MANASSE, M. S., MCGEOCH, L. A., AND OWICKI, S. Competitive randomized algorithms for non-uniform problems. In *Symposium on Discrete Algorithms* (1990), pp. 301–309.
- [19] LARKBY-LAHET, J., SANTHANAKRISHNAN, G., AMER, A., AND CHRYSANTHIS, P. K. Step: Self-tuning energy-safe predictors. 125–133.
- [20] LU, Y.-H., MICHELI, G. D., AND BENINI, L. Requester-aware power reduction. In *Proceedings of the International Symposium on System Synthesis* (2000), pp. 18–24.
- [21] NIGHTINGALE, E. B., AND FLINN, J. Energy efficiency and storage flexibility in the blue file system. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).
- [22] PINHEIRO, E., BIANCHINI, R., CARRERA, E. V., AND HEATH, T. Load balancing and unbalancing for power and performance in cluster-based systems. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power* (September 2001).
- [23] QIU, Q., AND PEDRAM, M. Dynamic power management based on continuous-time markov decision processes. In *Proceedings of the Design Automation Conference* (New Orleans, LA, USA, June 1999), pp. 555–561.
- [24] RYBCZYNSKI, J. P., LONG, D. D. E., AND AMER, A. Expecting the unexpected: adaptation for predictive energy conservation. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability* (New York, NY, USA, 2005), ACM Press, pp. 130–134.
- [25] SAMSUNG. Samsung teams with microsoft to develop first hybrid hard drive with nand flash memory, 2005.
- [26] SIMUNIC, T., BENINI, L., GLYNN, P., AND MICHELI, G. D. Dynamic power management for portable systems. In *Proceedings of the International Conference on Mobile Computing and Networking* (2000), pp. 11–19.
- [27] SRIVASTAVA, M. B., CHANDRAKASAN, A. P., AND BRODERSEN, R. W. Predictive system shutdown and other architecture techniques for energy efficient programmable computation. *IEEE Transactions on VLSI Systems* 4, 1 (March 1996), 42–55.
- [28] WEISSEL, A., AND BELLOSA, F. Self-learning hard disk power management for mobile devices. In *Proceedings of the Second International Workshop on Software Support for Portable Storage (IWSSPS 2006)* (Seoul, Korea, Oct. 2006), pp. 33–40.
- [29] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O—a novel I/O semantics for energy-aware applications. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation* (December 2002).

Optimizing TCP Receive Performance

Aravind Menon and Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL

Abstract

The performance of receive side TCP processing has traditionally been dominated by the cost of the ‘per-byte’ operations, such as data copying and checksumming. We show that architectural trends in modern processors, in particular aggressive prefetching, have resulted in a fundamental shift in the relative overheads of per-byte and per-packet operations in TCP receive processing, making per-packet operations the dominant source of overhead.

Motivated by this architectural trend, we present two optimizations, receive aggregation and acknowledgment offload, that improve the receive side TCP performance by reducing the number of packets that need to be processed by the TCP/IP stack. Our optimizations are similar in spirit to the use of TCP Segment Offload (TSO) for improving transmit side performance, but without need for hardware support. With these optimizations, we demonstrate performance improvements of 45-67% for receive processing in native Linux, and of 86% for receive processing in a Linux guest operating system running on Xen.

1 Introduction

There is a large body of research on techniques to improve the performance of the TCP stack. Although a number of techniques have been proposed for improving the performance of the transmit side in TCP, such as zero-copy transmit and segmentation offload, there has been relatively little work on improving the receive side performance. New applications, such as storage area networks, make receive performance a possible concern. In this paper we analyze the receive side performance of TCP on modern machine architectures, and we present new mechanisms to improve it.

Conventionally, the dominant source of overhead in TCP receive processing has been the cost of the ‘per-byte’ operations, those operations that touch all bytes

of input data, such as data copying and checksumming [3, 9, 6, 12]. In contrast, the ‘per-packet’ operations, that are proportional to the number of packets processed, such as header processing and buffer management, were shown to be relatively cheap. Thus, unlike in TCP transmit processing, in TCP receive processing, there has been little emphasis on reducing the per-packet overheads.

The high cost of the per-byte operations, at least in older architectures, resulted in part from the fact that the network interface card (NIC) places newly arrived packets in main memory. All operations that touch the data of the packet thus incur compulsory cache misses. Modern processors, however, use aggressive prefetching to reduce the cost of sequential main memory access. Prefetching has a significant impact on the relative overheads of the per-byte and per-packet operations of TCP receive processing. Since the per-byte operations, such as data copying and checksumming, access the packet data in a sequential manner, their cost is much reduced by prefetching. In contrast, the per-packet operations, which access main memory in a non-sequential, random access pattern, do not benefit much. Thus, as the per-byte operations become cheaper, the per-packet overheads of receive side TCP processing become the dominant performance component.

Motivated by these architectural trends in modern processors, we present two optimizations to the TCP receive stack that focus on reducing the per-packet overheads. Our optimizations are similar in spirit to the TCP Segment Offload (TSO) optimization used for improving TCP transmit side performance. Unlike TSO, however, which requires support from the NIC, our optimizations can be implemented completely in software, and are therefore independent of the NIC hardware.

The first optimization is to perform packet ‘aggregation’: Multiple incoming network packets for the same TCP connection are aggregated into a single large packet, before being processed by the TCP stack. The cost of

packet aggregation is much lower than the gain achieved as a result of the TCP stack having to process fewer packets. TCP header processing is still done on a per-packet basis, because it is necessary for aggregation, but this is only a small part of the per-packet overhead. The more expensive components, in particular the buffer management, are executed once per aggregated packet rather than once per network packet, hence leading to a considerable overall reduction in cost.

In addition to aggregating received packets, we present a second optimization, TCP acknowledgment offload: Instead of generating a sequence of acknowledgment packets, the TCP stack instead generates a single TCP acknowledgement packet template, which is then turned into multiple TCP acknowledgment packets below the TCP stack. The gain in performance comes from the reduction in the number of acknowledgment packets to the processed by the transmit side of the TCP stack.

While the two optimizations are logically independent, receive aggregation creates the necessary condition for TCP acknowledgement offload to be effective, namely the need to send in short succession a substantial number of near-identical TCP acknowledgment packets.

We have implemented these optimizations for two different systems, a native Linux operating system, and a Linux guest operating system running on the Xen virtual machine monitor. We demonstrate significant performance gains for data-intensive TCP receive workloads. We achieve performance improvement of 45-67% in native Linux, and 86% in a Linux guest operating system running on Xen. Our optimizations also scale well with the number of concurrent receive connections, performing at least 40% better than the baseline system. Our optimizations have no significant impact on latency-critical workloads, or on workloads with small receive message sizes.

The organization of the rest of the paper is as follows. In section 2, we analyze the receive side processing overhead of TCP in native Linux and in a Linux guest operating system running on Xen, and show the significance of per-packet overheads in both settings. We present our optimizations to the TCP receive stack in sections 3 and 4. We evaluate our optimized TCP stack in section 5, and discuss related work in section 6. We conclude in section 7.

2 Background

We start by studying the impact of aggressive prefetching used by modern processors on the relative overheads of per-packet and per-byte operations in TCP receive processing. We profile the execution of the TCP receive stack in three different systems, a Linux uniprocessor system, a Linux SMP system, and a Linux guest operat-

ing system running on the Xen virtual machine monitor (VMM). We show that in all three systems the per-packet overheads have become the dominant performance component.

We next analyze the per-packet overheads on the receive path in more detail. We show that TCP/IP header processing is only a small fraction of the overall per-packet overhead. The bulk of the per-packet overhead stems from other operations such as buffer management. This observation is significant because TCP/IP header processing is the only component of the per-packet overhead that must, by necessity, be executed for each incoming network packet. The other operations are currently implemented on a per-packet basis, but need not be. The bulk of the per-packet overhead can therefore be eliminated by aggregating network packets before they incur the expensive operations in the TCP stack.

In the experiments in this section, we focus on analyzing bulk data receive workloads, in which all network packets received are of MTU size (1500 bytes for Ethernet). We use a simple netperf [1] like microbenchmark, which receives data continuously over a single TCP connection at Gigabit rate. Profiling provides us with the breakdown of the execution time across the different subsystems and routines.

The experiments are run on a 3.80 GHz Intel Xeon dual-core machine, with an Intel e1000 Gigabit NIC. The native Linux kernel version used is Linux 2.6.16.34, and for the virtual machine experiments we use Linux 2.6.16.38 running on Xen-3.0.4. Profile statistics are collected and reported using the OProfile [2] tool.

2.1 Impact of Prefetching

To counter the growing gap between processor and main memory speeds, modern processors use aggressive prefetching. The impact of this architectural trend on TCP receive processing is that per-packet overheads become dominant, while the per-byte overheads become less important. This can be seen from figure 1, which shows the breakdown of TCP receive processing overhead in a native Linux system, as a function of the extent of prefetching enabled in the CPU.

The total processing overhead is divided into three categories: the per-byte data copying routines, *per-byte*, the per-packet routines, *per-packet*, and other miscellaneous routines, *misc*. The *misc* routines are those that are not really related to receive processing, or cannot be classified as either per-packet or per-byte, for example scheduling routines. The per-packet routines include the device driver routines along with the TCP stack operations.

The three groups of histograms in figure 1 show the breakdown of the receive processing overhead for dif-

ferent CPU configurations: The *None* configuration uses no prefetching, *Partial* uses adjacent cache-line prefetching, and *Full* uses adjacent cache-line prefetching and stride-based prefetching.

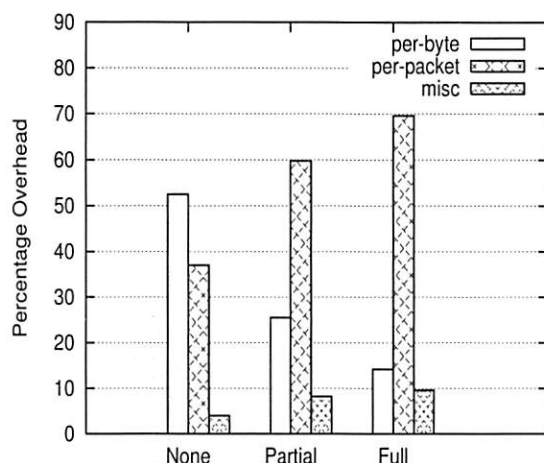


Figure 1: Impact of Prefetching on the Relative Cost of Per-Byte and Per-Packet Operations in TCP Receive Processing on a Uniprocessor as a Function of the Degree of Prefetching

As the CPU is configured to prefetch more aggressively, the contribution of the per-byte operations to the overall overhead declines from 52% to 14%. The proportion of the per-packet operations in the overall overhead increases correspondingly from 37% to nearly 70%, and becomes more important than the per-byte overheads.

The increase in importance of the per-packet costs is a consequence of the architectural evolution of microprocessors to cope with the increasing gap between memory and CPU speeds. This evolution is present across different architectures and operating systems, and is likely to further increase with future processors.

Figure 2 shows the relative overhead of per-packet and per-byte operations for three different systems, all with aggressive prefetching enabled: a Linux uniprocessor system (UP), a Linux SMP system (SMP), and a Linux guest operating system running on the Xen VMM (Xen). As can be clearly seen, in all three systems, per-packet overheads far outweigh the per-byte overheads.

2.2 Per-Packet Overhead

We now look in more detail into the per-packet overheads of receive processing. The key objective here is to distinguish between the operations that are a necessary part of TCP receive processing and have to be done on a per-packet basis, such as TCP/IP header processing, and operations that are not central to receive processing

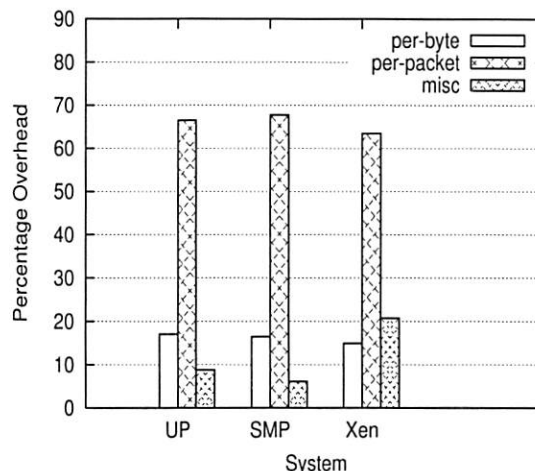


Figure 2: Per-byte vs. Per-packet Overhead in Uniprocessor, Multiprocessor and Virtualized Systems (with Full Prefetching Enabled)

or do not need to be executed on a per-packet basis, although for historical or architectural reasons they are implemented that way.

Figure 3 shows the breakdown of the TCP receive processing overhead into different kernel categories on Linux version 2.6.26.34, running on a uniprocessor 3 Ghz Xeon, with full prefetching enabled. The Y axis shows the number of busy CPU cycles per packet spent in the different routines. The X axis shows the different categories of overhead during receive processing.

The *per-byte* routines correspond to the per-byte operations in the receive path. The *misc* routines are those which are unrelated to receive processing, and are not strictly per-packet or per-byte. The per-packet overheads are split into five subgroups, *rx*, *tx*, *buffer*, *non-proto*, and *driver*, defined as follows:

1. **rx:** TCP/IP protocol processing routines on the receive path of the TCP stack.
2. **tx:** TCP/IP protocol processing routines on the transmit path of the TCP stack for transmission of ACKs.
3. **buffer:** Buffer management routines for network packets, ACK packets, and `sk_buffs`, the buffer metadata structure used in Linux.
4. **non-proto:** Other kernel routines which operate on per-packet basis, but are not part of the core TCP/IP protocol processing. Some of these are Linux-specific, such as routines for packet movement between `softirq` and `interrupt` context, whereas others are more generic, such as the packet filtering and network bridging routines.

5. **driver:** Device driver routines and routines running in interrupt mode.

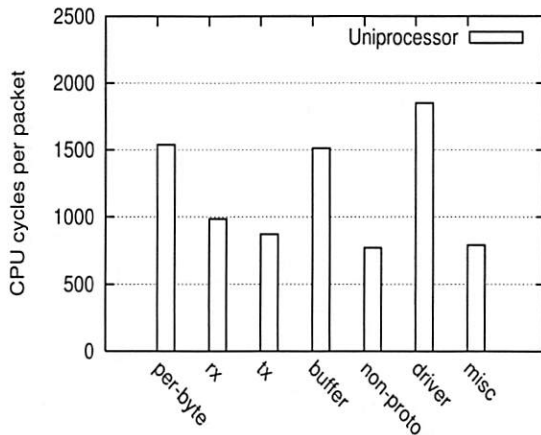


Figure 3: Breakdown of Receive Processing Overheads in a Uniprocessor system

The overhead of the device driver routines, *driver*, is roughly 21%. This overhead is per-packet, but that cannot be changed without modifications to the NIC. Thus, we henceforth distinguish between the *driver* routines and the other per-packet routines in the network stack, namely *rx*, *tx*, *buffer* and *non-proto*. Even excluding the *driver*, the overhead of these per-packet routines is 46%, which is much higher than the per-byte overhead of copying, 17%.

The overhead of the TCP/IP protocol processing itself, consisting of *rx* and *tx*, is only around 21% of the total overhead. The larger part of the per-packet overhead, around 25%, comes from the buffer management (*buffer*) and non-protocol processing related routines (*non-proto*) involved in handling of packets within the TCP/IP stack. Detailed profiling shows that most of the buffer management overhead is incurred in the memory management of *sk_buffs*, and not in the management of the network packet buffer itself. In conclusion, the bulk of the per-packet overhead in the network stack is incurred in routines which are not related to the protocol processing.

2.3 SMP Overheads

We now look in more detail at the per-packet overheads in an SMP environment. The system used is a Linux 2.6.16.34 SMP kernel running on dual-core Intel 3.0 GHz Xeon machine. Figure 4 shows the breakdown of the receive processing overheads in the SMP environment. Since the profile of the processing overhead on the

SMP is very similar to that observed on the uniprocessor system, we present both profiles in the same figure to illustrate the impact of multiprocessing on the different components of the TCP stack.

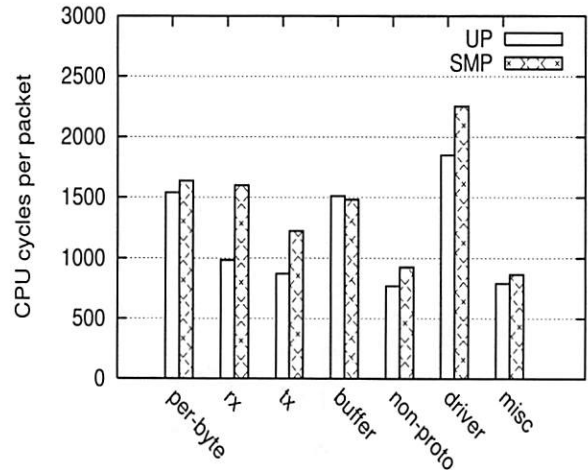


Figure 4: Breakdown of Receive Processing Overheads in an SMP vs. a UP environment

First, as for the uniprocessor system, the overhead of the per-packet routines (*rx*, *tx*, *buffer* and *non-proto*) is much higher (48%) than the per-byte copy overhead (16%). Second, in going from the uniprocessor to the SMP configuration, the per-byte copying overheads remain roughly the same, but there is a non-trivial increase in the overhead of some per-packet operations. In particular, in the SMP configuration the TCP receive routines (*rx*) incur 62%, and the TCP transmit routines (*tx*) incur 40% more overhead compared to the uniprocessor configuration. The buffer management routines, *buffer*, do not show significant difference in the two configurations.

The main reason for the increase in the overhead of the per-packet network stack routines is that in an SMP environment the TCP stack uses locking primitives to ensure safe concurrent execution. On the x86 architecture, locking is implemented through the use of lock-prefixed atomic *read-modify-write* instructions, which are known to suffer from poor performance on the Intel x86 CPUs.

In contrast, the per-byte data copy operations in the TCP stack can be implemented in a lock-free manner, and thus do not suffer from SMP scaling overheads. The buffer management routines do not suffer synchronization overheads because they are implemented in a mostly lock-free manner in Linux.

Thus, the locking and synchronization overheads of the TCP stack in an SMP environment are primarily in-

curred in the per-packet operations, and they grow in proportion to the number of packets that the TCP stack has to process. In conclusion, here again, mechanisms that reduce the number of packets handled by the TCP stack reduce these overheads.

2.4 Virtualization Overheads

We now analyze the receive processing overheads of the TCP stack in a virtual machine environment. In a virtual machine environment, device virtualization is an important part of the network I/O stack. We therefore analyze the extended network stack, including the virtualization stack.

Figure 5 shows a high level picture of the network virtualization architecture in Xen. Guest operating systems (guest *domains* in Xen) make use of a *virtual* network interface for network I/O, and do not access the *physical* network interface (NIC) directly. The *Driver* domain is a privileged domain that manages the physical network interface and multiplexes access to it among the guest domains. The *virtual* interface of the guest domain is connected to the physical NIC through a pair of *backend-frontend* paravirtualized drivers, and a network bridge in the driver domain. A more detailed description of the Xen networking architecture can be found in [7].

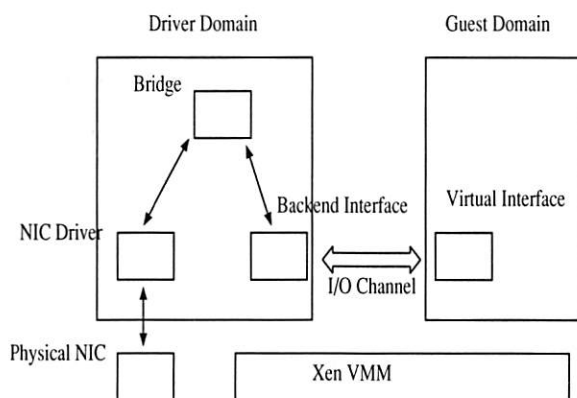


Figure 5: Xen I/O Architecture

Figure 6 shows the breakdown of the receive processing overhead for a Linux 2.6.16.38 guest domain running on Xen 3.0.4. The processing overheads in the guest domain, driver domain and Xen are split into the following categories:

1. **per-byte:** Data copy routines. This includes the two data copies on the receive path: the first from the driver domain into the guest domain, and second copy from the guest kernel into the guest application.

2. **non-*proto*:** This includes the bridge and netfilter routines in the driver domain, and similar non-protocol processing related routines in the guest domain. The bridge transfers packets received from the physical NIC to the backend interface. The overhead in this category is essentially a per-packet overhead.
3. **netback:** The netback driver initializes transfer of packets from the driver to the guest domain. Its overhead is mostly per-packet, and is proportional to the number of packet fragments it transfers.
4. **netfront:** The netfront driver receives packets from the driver domain and passes it onto the TCP stack. Its overhead is similar to the netback driver, and is proportional to the number of packet fragments it accepts.
5. **tcp rx and tx:** The transmit and receive TCP routines in the guest domain. These are per-packet overheads.
6. **buffer:** Buffer management routines, in both the driver domain and the guest domain. This is per-packet overhead.
7. **driver:** Device driver running in the Driver domain. This is a per-packet overhead.
8. **xen:** Xen manages domain scheduling, inter-domain interrupts, validation of packet transfer rights, etc. Its overhead cannot be classified strictly as either per-packet or per-byte.
9. **misc:** Other routines. These cannot be classified as either per-packet or per-byte.

The overall overhead of the per-packet routines in the receive path, comprising of the *non-*proto**, *netback*, *netfront*, *tcp rx*, *tcp tx* and *buffer* routines, adds up to roughly 56% of the total overhead, and is significantly higher than the per-byte copy overhead, 14%. This is in spite of the fact that there are two data copies involved, one from the driver domain to the guest domain, and the second from the guest kernel to the guest application.

The major part of the per-packet overhead is incurred in the routines of the network virtualization stack, and only a small part of the per-packet overhead is incurred in the TCP protocol processing in the guest domain. Thus, the *non-*proto** routines, the *netback* and *netfront* drivers, and the *buffer* management routines add up to 46% of the total overhead, whereas TCP/IP processing incurs only 10% of the total overhead.

Thus, here also, mechanisms that reduce the number of packets that need to be processed by the network stack

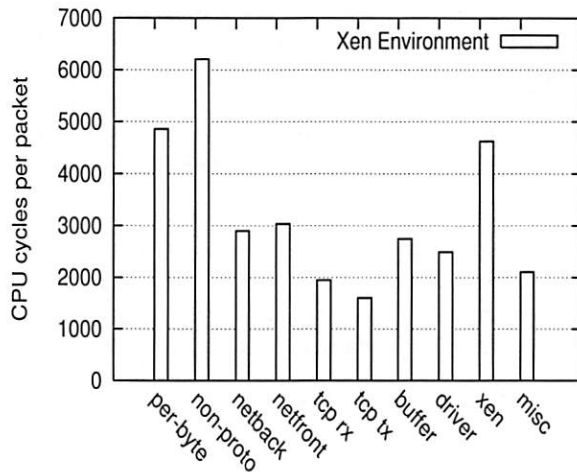


Figure 6: Breakdown of Receive Processing Overheads in Xen

can significantly reduce the overhead on the network virtualization path of guest domains.

2.5 Summary

First, architectural trends in microprocessors have resulted in a fundamental shift in the relative overheads of sequential and non-sequential memory access. Sequential memory access can be optimized by prefetching, while non-sequential access cannot. This has resulted in a fundamental shift in the relative overheads of per-byte and per-packet operations in TCP receive processing. While previously data copying was the primary performance bottleneck, in current systems the per-packet operations in the TCP receive path are the dominant overhead component. We have validated this trend for three different systems.

Second, reducing the number of packets to be processed by the TCP receive path holds promise as a solution for reducing the overhead of the per-packet operations. TCP/IP processing is the only overhead that needs to be incurred on a per-packet basis, but it is only a small component of the total overhead. The larger components of the overhead (system, virtualization, and SMP scaling) currently are incurred on a per-packet basis, but need not be.

We next present two optimizations to the TCP/IP stack that exploit these observations, Receive Aggregation and Acknowledgment Offload.

3 Receive Aggregation

The objective of Receive Aggregation is to reduce the number of packets that the network stack has to process on the receive path, while still ensuring that the TCP/IP protocol processing of the packets is done correctly.

The basic idea of Receive Aggregation is that, instead of allowing the network stack to process packets received from the NIC directly, the packets are preprocessed and coalesced into ‘aggregated’ TCP packets, which are then passed on to the network stack for further processing. Multiple ‘network’ TCP packets are aggregated into a single ‘host’ TCP packet, thereby reducing the number of packets that the network stack has to process.

Ideally, aggregation would be done entirely in a proxy between the driver and the TCP stack, and without any changes to the rest of the kernel code. For correctness and performance, this complete separation cannot be achieved, and small changes to the driver and the TCP layer are necessary. No changes were made to the IP layer or the layout of the kernel data structure for storing packets, `sk_buff` in Linux.

3.1 Which Packets are Aggregated

Receive Aggregation takes place at the entry point of the network stack. Packet coalescing is done for network TCP packets which arrive “in-sequence” on the same TCP connection. Thus, the incoming packets must have the same source IP, destination IP, source port, and destination port fields. The packets must also be in sequence, both by TCP sequence number and by TCP acknowledgment number. Thus, the sum of the TCP sequence number of one packet and its length must be equal to the TCP sequence number of the next packet. Also, a TCP packet later in the aggregated sequence must have a TCP acknowledgment number greater than or equal to that of a previous packet in the sequence.

Packet aggregation is done only for valid TCP packets, i.e., those with a valid TCP and IP checksum. We verify only the IP checksum field of the network TCP packet before it is used for aggregation. For the TCP checksum, we assume the common case that the NIC supports checksum offloading, and has validated the TCP checksum. This is because verifying the TCP checksum in software would make the aggregation expensive.

If the network card does not support receive checksum offloading, we do not perform Receive Aggregation.

TCP packets of zero length, such as pure ACK packets, are not aggregated. This simplifies the handling of duplicate ACKs in the TCP layer, and is discussed in section 3.6.

Since the TCP and IP headers support a large number of option fields, it is not possible to aggregate two

TCP packets if they contain different option fields. Also, the aggregation function becomes quite complicated if it has to support all possible TCP and IP options. Thus, for simplicity, we only aggregate TCP packets whose IP headers do not use any IP options or IP fragmentation, and whose TCP headers use only the TCP timestamp option.

Packets which fail to match any of the conditions for Receive Aggregation are passed unmodified to the network stack. In doing so, we ensure that there is no packet reordering between packets of the same TCP connection, i.e., any partially aggregated packet belonging to a TCP connection is delivered before any subsequent unaggregated packet is delivered.

3.2 Aggregated Packet Structure

Once a valid set of network packets is identified for aggregation, according to the conditions described above, the aggregation function coalesces them into an aggregated TCP packet for the network stack to process.

The aggregated TCP packet is created by ‘chaining’ together individual TCP packets to form the ‘fragments’ of the aggregated packet, and by rewriting the TCP/IP header of the aggregated packet. The TCP/IP header of the first TCP fragment in the chain retains its becomes the header of the aggregated packet, while the subsequent TCP fragments retain only their payload. The chaining is done in an OS specific manner. In Linux, for instance, chaining is done by setting the fragment pointers in the `sk_buff` structure to point to the payload of the TCP fragments. Thus, there is no data copy involved in packet aggregation.

The TCP/IP header of the aggregated packet is rewritten to reflect the packet coalescing. The IP packet length field is set to the length of the total TCP payload (comprising all fragments) plus the length of the header. The TCP sequence number field is set to the TCP sequence number of the first TCP fragment, and the TCP acknowledgment number field is set to the acknowledgment number of the last TCP fragment. The TCP advertised window size is set to the window size advertised in the last TCP fragment. A new IP checksum is calculated for the aggregated packet using its IP header and the new TCP pseudo-header. The TCP checksum is not recomputed (since this would be expensive), instead we indicate that the packet checksum has been verified by the NIC.

The TCP timestamp in the aggregated packet is copied from the timestamp in the last TCP fragment of the aggregated packet. Theoretically, this results in the loss of timestamp information, and may affect the precise estimation of RTT values. However, in practice, since only packets which arrive very close in time to each other are aggregated, the timestamp values on all the TCP frag-

ments are expected to be the same, and there is no loss of precision. We give a more rigorous argument for this claim in section 3.6.

Finally, the aggregated TCP packet is augmented with information about its constituent TCP fragments. Specifically, the TCP acknowledgment number of each TCP fragment is saved in the packet metadata structure (`sk_buff`, in the case of Linux). This information is later used by the TCP layer for correct protocol processing.

3.3 When Aggregation Stops

Network Packets are aggregated as they are received from the NIC driver. The maximum number of network TCP packets that get coalesced into an aggregated TCP packet is called the Aggregation Limit. Once an aggregated packet reaches the Aggregation Limit, it is passed on to the network stack.

The actual number of network packets that get coalesced into an aggregated packet may be smaller than the Aggregation Limit, and depends on the network workload and the arrival rate of the packets. If an aggregated packet contains less than the Aggregation Limit number of network packets, and no more network packets are available for processing, then this semi-aggregated packet is passed on to the network stack without further delay. Thus, the network stack is never allowed to remain idle while there are packets to process in the system. Receive Aggregation is thus work-conserving and does not add to the delay of packet processing.

We expect the performance benefits of Receive Aggregation to be proportional to the number of network packets coalesced into an aggregated packet. However, the incremental performance benefits of aggregation are expected to be marginal beyond a certain number of packets. Thus, the Aggregation Limit serves as an upper bound on the maximum number of packets to aggregate, and should be set to a reasonable value at which most of the benefits of aggregation are achieved. We determine a good cut-off value for the Aggregation Limit experimentally.

3.4 Modifications to the TCP layer

The aggregated TCP packet is received by the network stack, and gets processed through the MAC and IP layers in the same way as a regular TCP packet is processed. However, at the TCP layer, modifications are required in order to handle aggregated packets correctly. This is because TCP protocol processing in the TCP layer is dependent on the actual number of ‘network’ TCP packets, and on the exact sequence of TCP acknowledgments received. Since Receive Aggregation modifies both these

values, changes are required to the TCP layer to do correct protocol processing for aggregated packets.

There are two specific situations for which the TCP processing needs to be modified:

1. **Congestion Control:** The congestion window of a TCP sender is updated based on the *number* of TCP Acknowledgment packets received by the sender, and not on the total number of *bytes* acknowledged by the receiver. Since Receive Aggregation sets the TCP ACK field in the aggregated packet to the ACK field of the last TCP fragment, the conventional TCP layer implementation would set the congestion window differently from what is expected in the absence of receive aggregation.

The modified TCP layer computes the congestion window using the TCP acknowledgment numbers of all the TCP fragments of the aggregated TCP packet, instead of just using the final acknowledgment number. As noted in section 3.1, the acknowledgment numbers of the individual TCP fragments are stored in the packet metadata structure (`sk_buff`) when Receive Aggregation is performed.

2. **TCP Acknowledgments:** The TCP protocol specifies that an acknowledgment packet must be generated for every alternate full TCP segment received by the receiver. Since Receive Aggregation coalesces multiple network TCP packets into a single aggregated packet, the conventional TCP layer would conclude that it has received only a single TCP segment, and therefore generate the wrong number of acknowledgment packets.

The modified TCP layer computes the correct number of acknowledgments by taking into account the individual TCP fragments, instead of considering the whole aggregated packet as one segment. This information is also stored in the `sk_buff` structure during receive aggregation.

3.5 Implementation

Receive Aggregation is implemented at the entry point of the network stack processing routines. For the Linux network stack, this is the entry point of the `softirq` for receive network processing.

The network card driver, which receives packets from the NIC, is modified to enqueue the received packets into a special producer-consumer 'aggregation queue'. The Receive Aggregation routine, running in `softirq` context, 'consumes' the packets dropped into the queue and processes them for aggregation. The 'aggregation queue' is a per-CPU queue, and is implemented in a

lock-free manner. Thus, there is no locking overhead incurred for accessing this queue concurrently between different CPUs, or between the `interrupt` context and the `softirq` context.

The packets dropped into the aggregation queue by the NIC driver are 'raw' packets, i.e., they are not encapsulated in the Linux socket buffer metadata structure, `sk_buff`. The reason for this, as discussed in section 2.2, is that memory management of `sk_buffs` is a significant part of the buffer management overhead of network packets. We avoid this overhead by allocating the `sk_buff` only for the final aggregated packet, in the Receive Aggregation routine. For Linux drivers, this also allows us to avoid the MAC header processing of network packets in the driver, which is moved to the Receive Aggregation routine.

Packets are consumed from the aggregation queue by the aggregation routine and are hashed into a small lookup table, which maintains a set of partially aggregated TCP packets. If the new network TCP packet 'matches' a previously hashed packet, i.e., it can be coalesced with this packet (based on the conditions described in 3.1), then the two are aggregated. Otherwise, the partially aggregated packet is delivered to the network stack, and the new packet is saved in the lookup table.

Packets delivered to the network stack are processed synchronously, and thus control returns to the aggregation routine only when the network stack is idle. Thus, in order to remain work-conserving, whenever the aggregation routine runs out of network packets to process (i.e., the aggregation queue is empty), it immediately clears out all partially aggregated packets in the lookup table, and delivers them to the network stack. This ensures that packets do not remain waiting for aggregation, while the network stack is idle.

3.6 Correctness

Receive Aggregation is done only for a restricted set of in-order TCP packets which the TCP layer 'expects' while it is in error-free mode of operation. Any TCP packet requiring special handling by the TCP layer, off the common path, is passed on to the stack without aggregation, and thus is handled correctly by the TCP layer. Thus, all the error-handling and special case handling of packets in the TCP layer works correctly. We give a few examples below:

1. **Duplicate or Out-of-order packets:** Since these packets are not in correct sequence (by TCP sequence number), they are not aggregated and are handled directly by the TCP layer.

2. Selective TCP ACKs: Since TCP options other than the timestamp option are not handled by aggregation, TCP packets with selective ACKs are passed unmodified.
3. Duplicate ACKs: A duplicate ACK packet does not contain data in its payload. Since pure ACKs are never aggregated, these are handled correctly by the TCP layer.

We now explain why using timestamps from only the last TCP fragment in the aggregated packet does not result in lack of precision. At gigabit transmit rate, a single TCP sender machine can transmit packets at the rate of roughly 81,000 packets per second. The precision of the timestamp value itself, however, is typically 10 ms (if the system uses a 100 Hz clock), or 1 ms at best (with a 1000 HZ clock). Thus, roughly every 80 consecutive packets transmitted by a sender are expected to have the same timestamp to begin with. Since Receive Aggregation coalesces together packets which arrive very close to each other in time, we expect these packets to already have the same timestamp value.

4 Acknowledgment Offload

Our second optimization for reducing the per-packet overhead of receive processing is Acknowledgment Offload. Acknowledgment Offload reduces the number of TCP ACK packets that need to be processed on the transmit path of receive processing, and thus reduces the overall per-packet overhead.

TCP acknowledgment packets constitute a significant part of the overhead of TCP receive processing. This is because, in the TCP protocol, one TCP ACK packet must be generated for every two full TCP packets received from the network. Thus, TCP ACK packets constitute at least a third of the total number of packets processed by the network stack. Since the overhead of the network stack is predominantly per-packet, reducing the TCP ACK transmission overhead is essential for achieving good receive performance.

4.1 Basic Idea

Acknowledgment Offload allows the TCP layer to combine together the transmission of consecutive TCP ACK packets of the same TCP connection into a single 'template' ACK packet.

To transmit the successive TCP ACK packets, the TCP layer creates a 'template' TCP ACK packet representing the individual ACK packets. The template ACK packet is sent down the network stack like a regular TCP packet. On reaching the NIC driver (or a proxy for the driver), the

individual TCP ACK packets are re-generated from the template ACK packet, and are sent out on the network.

4.2 Template ACK packet

The template ACK packet for a sequence of consecutive ACKs is represented by the first ACK packet in the sequence, along with the ACK sequence numbers for the subsequent ACK packets, which is stored in the template packet's metadata structure (`sk_buff` in Linux).

The TCP and IP headers of the successive ACK packets of a TCP connection share most of the fields of the header. In particular, only the ACK sequence number and the IP checksum field differ between the successive packets. (This is assuming they are generated sufficiently close in time, so that the TCP timestamps are identical). Thus, the information present in the template ACK packet is sufficient to generate the individual ACK packets in the sequence.

The NIC driver is modified to handle the template ACK packet differently. The template packet is not transmitted on the network directly. Instead, the driver makes the required number of copies for the network ACK packets. It then rewrites the ACK sequence number for the network packets, recomputes the TCP checksum, and transmits the sequence of TCP ACK packets on the NIC.

4.3 When it is used

Acknowledgment Offload is preferably used in conjunction with the Receive Aggregation optimization. This is because, in a conventional TCP stack, TCP ACK packets are generated and transmitted synchronously in response to received TCP packets (except for delayed TCP ACKs). Since the received TCP packets are also processed by the TCP stack synchronously, the TCP layer does not generate opportunities to batch together the generation of successive ACK packets.

However, with Receive Aggregation, an aggregated TCP packet effectively delivers multiple network TCP packets to the TCP layer simultaneously. This provides the TCP layer an opportunity to generate a sequence of consecutive TCP ACK packets simultaneously, and at this point, it can make use of Acknowledgment Offload to transmit the ACK packets.

5 Evaluation

We have implemented Receive Aggregation and Acknowledgment Offload in a stock 2.6.16.34 Linux kernel, and in the Xen VMM version 3.0.4 running Linux 2.6.16.38 guest operating systems.

We first evaluate the performance benefits of the receive optimizations for three systems: a uniprocessor

Linux system, an SMP Linux system, and a Linux guest operating system running on the Xen VMM. Next, we experimentally determine a good cut-off value for the Aggregation Limit. In the third set of experiments, we demonstrate the scalability of our system as we increase the number of concurrent receive processing connections. Finally, we demonstrate that our optimizations do not affect the performance of latency-sensitive workloads.

5.1 Performance Benefits

We use a receive microbenchmark to evaluate the TCP receive performance of the system under test. This microbenchmark is similar to the netperf [1] TCP streaming benchmark and measures the maximum TCP receive throughput which can be achieved over a single TCP connection.

The server machine used for our experiments is a 3.0 GHz Intel Xeon machine, with 800 MHz FSB and 512 MB of DDR2-400 memory. The machine is equipped with five Intel Pro1000 Gigabit Ethernet cards, fitted in 133 MHz, 64 bit PCI-X slots. We run one instance of the microbenchmark for each network card. The ‘receiver’ end of each microbenchmark instance is run on the server machine and the ‘sender’ end is run on another client machine, which is connected to the server machine through one of the Gigabit NICs. The sender continuously sends data to the receiver at the maximum possible rate, for the duration of the experiment (60s). The final throughput metric reported is the sum of the receive throughput achieved by all receiver instances.

Overall Results

Figure 7 shows the overall performance benefit of using Receive Aggregation and Acknowledgment Offload in the three systems. The figure compares the receive performance of the three systems (throughput, in Mb/s) with and without the use of the receive optimizations. The ‘Linux UP’ histograms show the performance for the uniprocessor Linux system, ‘Linux SMP’, for the SMP Linux system, and ‘Xen’, for the Linux guest operating system running on the Xen VMM.

The performance results for the three systems are as follows.

For the uniprocessor Linux system, the unmodified (Original) Linux TCP stack reaches full CPU saturation at a throughput of 3452 Mb/s. With the use of the receive optimizations, the system (Optimized) is able to saturate all the five Gigabit network links, to reach a throughput of 4660 Mb/s. The CPU is still not fully saturated at this point and is at 93% utilization. The system is thus constrained by the number of NICs, and with more NICs,

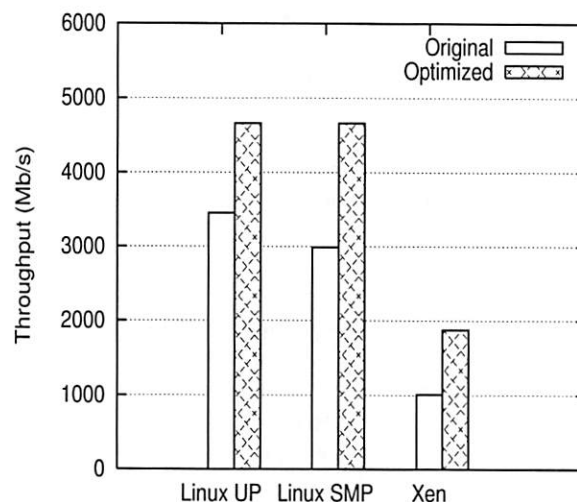


Figure 7: Overall Performance Improvement

it can theoretically reach a (CPU-scaled) throughput of 5050 Mb/s. The performance gain of the system is thus 35% in absolute units and 45% in CPU-scaled units.

For the SMP Linux system, the performance of the Original system is 2988 Mb/s, whereas the optimized system is able to saturate all five NICs to reach a throughput of 4660 Mb/s. As in the uniprocessor case, the optimized SMP system is still not CPU saturated and is at 93% CPU utilization. Thus, the performance gain in the SMP system is 55% in absolute terms and 67% in CPU-scaled units.

For the Linux guest operating system running on Xen, the unoptimized (Original) system reaches full CPU saturation with a throughput of only 1088 Mb/s. With the receive optimizations, the throughput is improved to 1877 Mb/s, which is 86% higher than the baseline performance.

The contribution of Acknowledgment Offload to the above performance improvements is non-trivial. Using just Receive Aggregation without Acknowledgment Offload, the performance improvement to the three configurations is, respectively, 26%, 36% and 45%, with CPU utilization reaching 100% in all three cases.

Analysis of the Results

We can better understand the performance benefits of the receive optimizations by comparing the overhead profiles of the network stack in the three configurations, with and without the use of the optimizations.

Figure 8 compares the performance overhead of the network stack for the uniprocessor Linux configuration, and shows the breakdown of the CPU cycles incurred per packet on the receive path. In addition to the different

per-packet and per-byte categories discussed in section 2, there is a new category *aggr*, which measures the overhead of doing Receive Aggregation. The overhead of Acknowledgment Offload itself is included as part of the device driver overhead.

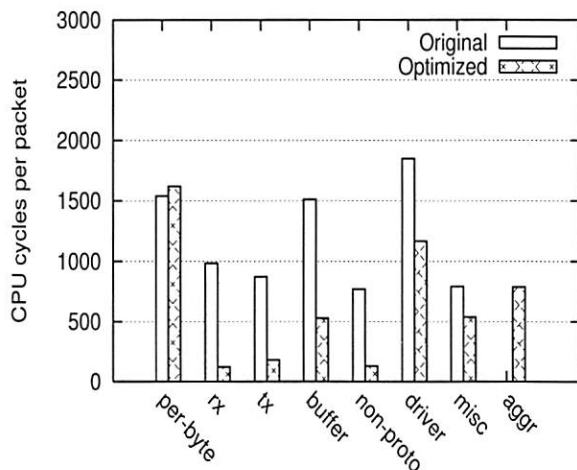


Figure 8: Receive processing overheads (UP)

Receive Aggregation and Acknowledgment Offload effectively reduce the number of packets processed in the network stack by a factor of up to 20, which is the Aggregation Limit on our system. This greatly reduces the overhead of all the per-packet components in the network stack. The total overhead of all per-packet components (*rx*, *tx*, *buffer* and *non-proto*) is reduced by factor of 4.3.

The main increase in overhead for the optimized network stack is the Receive Aggregation function itself (*aggr*). We note that the bulk of the overhead incurred for Receive Aggregation (789 cycles/packet) is due to the compulsory cache miss which is incurred in the early demultiplexing of the packet header. Since the device driver itself does not perform any MAC header processing in the optimized network stack, its overhead is reduced by 681 cycles/packet, since it avoids the compulsory cache miss.

Figure 9 shows the receive processing overhead for the optimized and unoptimized network stack in the SMP Linux configuration.

The overall trends in this figure are similar to those for the uniprocessor configuration. As in the uniprocessor case, the overhead of the per-packet routines, *rx*, *tx*, *buffer* and *non-proto* is greatly reduced. With the receive optimizations, the total overhead of all the per-packet components (*rx*, *tx*, *buffer* and *non-proto*) in the network stack is reduced by a factor of 5.5.

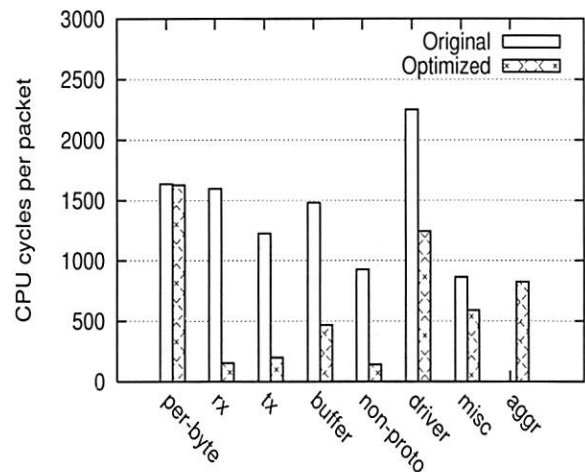


Figure 9: Receive processing overheads (SMP)

In the original SMP configuration, there is an increase in the overhead of the per-packet TCP routines relative to the uniprocessor case, because of the locking and synchronization overhead in the TCP stack. In the optimized network stack, both Receive Aggregation and Acknowledgment Offload are implemented in a CPU-local manner, and thus do not incur any additional synchronization overheads.

Finally, figure 10 shows the breakdown of the receive processing overhead with receive optimizations in the Linux guest operating system.

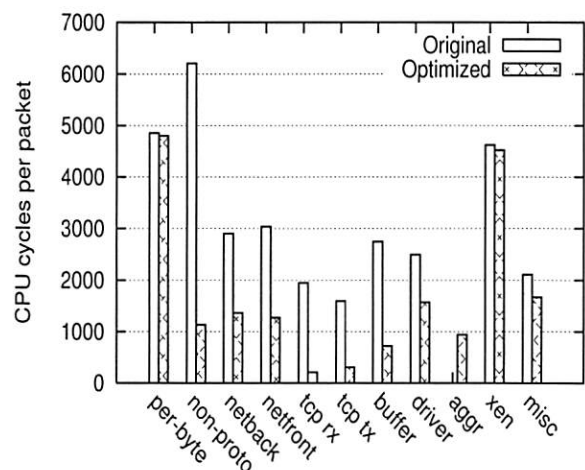


Figure 10: Receive processing overheads (Xen)

In the guest Linux configuration, the total overhead of the per-packet routines (*non-proto*, *netback*, *netfront*, *tcp rx*, *tcp tx* and *buffer*) in the

network virtualization stack is reduced by a factor of 3.7 with the use of the receive optimizations. The greatest visible reduction is in the overhead of the non-`proto` routines, which includes the bridging and netfilter routines in the driver domain and the guest domain. The overheads of the `netfront` and `netback` paravirtual drivers are reduced to a lesser extent, primarily because they incur a per-TCP fragment overhead instead of a purely per-packet overhead. Other per-packet components, such as the TCP receive and transmit routines (TCP `rx` and TCP `tx`) and buffer management (`buffer`) show similar reduction in overhead as in the native Linux configurations.

The overhead of Receive Aggregation (`aggr`) itself is small compared to the other overheads.

5.2 Choosing the Aggregation Limit

The performance benefit of Receive Aggregation is proportional to the number of TCP packets which are combined to create the aggregated host TCP packet. A greater degree of aggregation results in a greater reduction in the per-packet overhead. However, beyond a limit, packet aggregation does not yield further benefits. We determine a good cut-off value for this Aggregation Limit experimentally.

Figure 11 shows the total CPU execution overhead (in CPU cycles per packet) incurred for receive processing in a uniprocessor Linux system, as a function of Aggregation Limit.

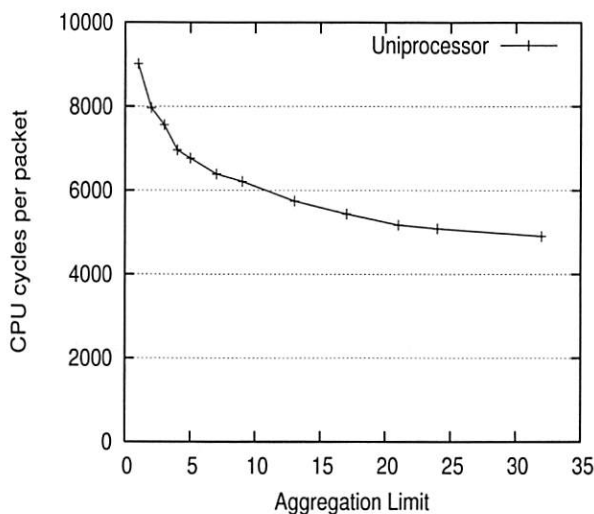


Figure 11: CPU overhead vs. Aggregation Limit

Increasing the Aggregation Limit initially yields a sharp reduction in the CPU processing overhead of packets. The figure shows that most of the benefits of Receive Aggregation can be achieved with a relatively small Ag-

gregation Limit. We choose a value of 20 for the Aggregation Limit as it can be seen that additional aggregation does not yield any substantial improvement.

The Aggregation Limit measured above can also be derived analytically, since it only depends on the percentage overhead of the per-packet operations whose overhead can be scaled down by aggregation. For instance, if $x\%$ of the overhead is constant, and $y\%$ is the per-packet overhead that can be reduced by aggregation (with $x + y = 100$), then using an aggregation factor of k should reduce the system CPU utilization from $x + y$ to $x + y/k$. Figure 11 appears to match the plot of $x + y/k$ as a function of k fairly well. This gives us confidence that the Aggregation Limit chosen is quite robust and not arbitrary, and it will hold across a number of different systems.

5.3 Scalability

The previous sections demonstrate the performance benefits of our optimizations when the workload consists of a small number of high-volume TCP connections. We now evaluate how the optimizations scale as we increase the number of concurrent TCP connections receiving data.

The benchmark we use is a multi-threaded version of the receive microbenchmark. We create a number of receiver threads, each of the threads running the receiver microbenchmark and connected to a different sender process. We measure the cumulative receive throughput as a function of the number of receive connections.

Figure 12 shows how the system scales as a function of the number of connections, both in the original and the optimized system. The figure compares a baseline 2.6.16.34 Linux SMP system (Original), with the optimized Linux system (Optimized).

The figure shows that our optimizations scale very well even as we increase the number of concurrent connections to 400, with the optimized system performing 40% better than the baseline system, at 400 connections. This demonstrates that Receive Aggregation is effective in reducing the number of packets even in the presence of concurrency.

5.4 Impact on Latency Sensitive Workloads

We use the `netperf` [1] TCP Request/Response benchmark to evaluate the impact of the receive optimizations on the latency of packet processing.

This benchmark measures the interactive request-response performance of a client and server program connected by a TCP connection. The client sends the server a one-byte 'request', and waits for a one-byte

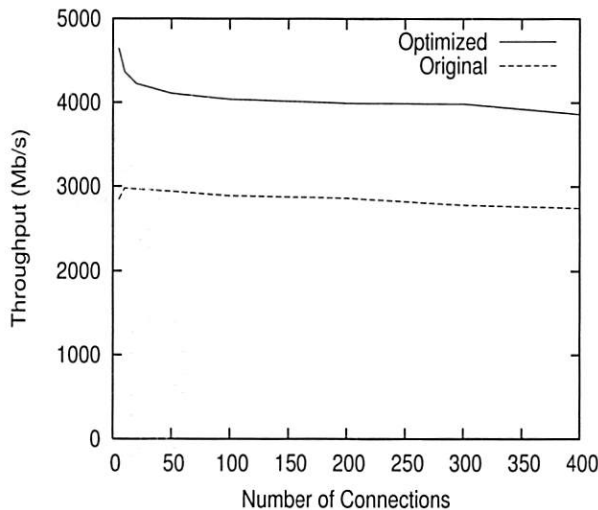


Figure 12: Scalability

	Requests/sec (Original)	(Optimized)
Linux UP	7874	7894
Linux SMP	7970	7985
Xen	6965	6953

Table 1: Impact of Receive Optimizations on Latency

‘response’ from the server. On receiving the response, the client immediately sends another request. The benchmark measures the maximum request-response rate achieved between the client and the server.

Table 1 compares the performance of the three systems on the TCP Request/Response benchmark. The table shows that our receive optimizations have no noticeable impact on the latency of packet processing in the network stack.

This is because of the work-conserving nature of Receive Aggregation. Since there is only one network packet to process at a time, no packet aggregation is done, and the packet is passed on to the network stack immediately to prevent it from being idle.

5.5 Discussion

The performance results presented in this section were achieved in a LAN environment, where the low network latencies and small inter-packet delays allow Receive Aggregation to effectively coalesce multiple consecutive TCP packets. The important condition for Receive Aggregation to work effectively is to have a sufficient number of consecutive TCP packets received within short interval of each other.

One example of a real-world situation where Receive Aggregation is applicable is a Storage Area Network

(SAN) using iSCSI, where storage servers have high bandwidth processing requirements for transferring (including receiving) large files. In general, data intensive workloads running in a LAN environment would gain the most from these optimizations.

Under other network conditions, the performance benefits of our optimizations may vary, depending on the degree of aggregation possible. However, the overall performance will never get worse than the original system. We verified this by setting the Aggregation Limit to one in our LAN experiments, which measures the overhead of our system in the absence of any aggregation. We observed no degradation in the performance relative to the baseline.

6 Related Work

Initial analysis of TCP performance [3] identified the per-byte data touching operations to be the major source of overhead for TCP. This led to the development of a number of techniques for avoiding data copy, both in software [15] [10], and hardware [13, 11]. Techniques such as zero-copy transmit and hardware checksum offload have now become common in modern network cards [12, 4, 6].

Later work [9] identified the per-packet overhead as the dominant source of overhead for real-world workloads, which are dominated by small message sizes. This led to the development of offloading techniques for reducing per-packet overheads, such as TCP segmentation offload.

Recently, some high end network cards have started providing more complex offload support for TCP receive processing, such as Large Receive Offload (LRO) in Neterion NICs [8]. The idea of LRO is similar to that of Receive Aggregation, except that it is performed in the NIC, and thus it can reduce the per-packet overhead incurred in the network driver. However, a pure-software approach such as Receive Aggregation is much more generic, and can yield much of the benefit of packet aggregation in a hardware independent manner. Additionally, the Neterion NIC does not support Acknowledgment Offload, and thus does not offer support for reducing the overhead on the ACK transmit path.

Jumbo frames, which allow the ethernet MTU size to be set to 9000 bytes, can also effectively help reduce the per-packet overheads for bulk data transfers. However, they require the whole LAN network to be upgraded to use the same MTU size. Receive Aggregation and Acknowledgment Offload are effective at improving the network stack performance irrespective of the network MTU size or networking hardware used.

Receive Aggregation requires TCP packets to be demultiplexed early on in the network stack. Similar early

demultiplexing mechanisms have been explored in the context of resource accounting in Lazy Receive Processing (LRP) [5]. LRP, however, does not yield any performance improvements.

The idea of Receive Aggregation is also similar to the idea of interrupt throttling supported by many network cards. Since interrupt processing is expensive, interrupt throttling prevents Operating Systems from spending too much time in processing interrupts [14]. Similarly, Receive Aggregation reduces the CPU overhead of TCP receive processing by reducing the number of host TCP packets that the network stack has to process.

7 Conclusions

In this paper, we showed that architectural trends in the evolution of microprocessors have shifted the dominant source of overhead in TCP receive processing from per-byte operations, such as data copy and checksumming, to the per-packet operations. Motivated by this architectural trend, we presented two optimizations to receive side TCP processing, Receive Aggregation and Acknowledgment Offload, which reduce its per-packet overhead. These optimizations result in significant improvements in the performance of TCP receive processing in native Linux (by 45-67%), and in virtual Linux guest operating systems running on the Xen VMM (by 86%).

8 Acknowledgments

We would like to thank Emmanuel Cecchet, Olivier Crameri, Simon Schubert, Nikola Knezevic and Kateřina Argyraki for discussions and useful feedback. This work was supported in part by the Swiss National Science Foundation grant number 200021-111900.

References

- [1] The netperf benchmark. <http://www.netperf.org/netperf/NetperfPage.html>.
- [2] Oprofile. <http://oprofile.sourceforge.net>.
- [3] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6), June 1989.
- [4] Intel Corporation. Small packet traffic performance optimization for 8255x and 8254x Ethernet Controllers. Technical Report application Note (AP-453), Sept 2003.
- [5] Peter Druschel and Gaurav Banga. Lazy Receive Processing (LRP): A Network Subsystem Architecture for Server Systems. In *OSDI*, 1996.
- [6] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. Tcp performance re-visited. In *International Symposium on Performance Analysis of Systems and Software, IPASS*, Austin, TX, 2003.
- [7] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Oct 2004.
- [8] Leonid Grossman. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Ottawa Linux Symposium*, Ottawa, 2005.
- [9] Jonathan Kay and Joseph Pasquale. Profiling and Reducing Processing Overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, December 1996.
- [10] H. Keng and J. Chu. Zero-copy TCP in Solaris. In *USENIX 1996 Annual Technical Conference*, 1996.
- [11] K. Kleinpaste, P. Steenkiste, and B. Zill. Software Support for Outboard Buffering and Checksumming. In *ACM SIGCOMM Symposium*, 1995.
- [12] Srihari Makineni and Ravi Iyer. Architectural characterization of TCP/IP packet processing on the Pentium M microprocessor. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.
- [13] Dave Minton, Greg Regnier, Jon Krueger, Ravishankar Iyer, and Srihari Makineni. Addressing TCP/IP Processing Challenges Using the IA and IXP Processors. *Intel Technology Journal*, November 2003.
- [14] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt driven kernel. In *ACM Transactions on Computer Systems*, 1997.
- [15] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.

ConfiDNS: Leveraging Scale and History to Detect Compromise

Lindsey Poole, Vivek S. Pai
Princeton University

Abstract

While cooperative DNS resolver systems, such as CoDNS, have demonstrated improved reliability and performance over standard approaches, their security has been weaker, since any corruption or misbehavior of a single resolver can easily propagate throughout the system.

We address this weakness in a new system called ConfDNS, which augments the cooperative lookup process with configurable policies that utilize multi-site agreement and per-site lookup histories. Not only does ConfDNS provide better security than cooperative approaches, but for up to 99.8% of unique lookups, ConfDNS exceeds the security of standard DNS resolvers. ConfDNS provides these benefits while retaining the other benefits of CoDNS, such as incremental deployability, higher reliability, and improved performance, in some cases faster than CoDNS.

1 Introduction

The use of distributed computing to address performance and reliability problems in the Domain Name System (DNS) [18] has recently received much research attention, and has spawned two widely-deployed distributed systems, CoDNS [19] and CoDoNS [22]. Both of these systems provide clients with improved reliability when performing DNS lookups by distributing the queries across nodes in the system. These systems fetch name-to-IP translations from the existing legacy DNS infrastructure as needed to provide an upgrade path for users.

Due to their interaction with vulnerable legacy DNS infrastructure, these systems can be less secure than traditional local DNS resolvers, even if these DNS replacements are written securely and use secure inter-node communication. If any node performs a DNS resolution and receives an incorrect answer, that answer can be propagated to other nodes. The incorrect answer can occur because of a failure or compromise at a local DNS resolver, or from UDP packet spoofing when the node tries to communicate with an external DNS server. Ironically, the use of aggressive caching and multi-hop request forwarding to improve resilience under flooding attacks can actually cause polluted responses to be kept longer and spread further, magnifying the scope of the problem when compared to traditional DNS configurations.

Rather than being a fundamental trade-off in trying to support legacy DNS while achieving better reliability,

we show that worse security is not an unavoidable by-product of cooperative DNS systems. For most DNS deployment scenarios, the greater scale of cooperative DNS systems can be leveraged to provide better security than legacy DNS resolvers for the vast majority of queries. Where scale cannot be used, observing the history of DNS queries can provide some assurance that DNS replies have not been modified. Between these two options, only a small fraction of DNS queries need to trade security for reliability or performance.

In this paper, we present ConfDNS, a cooperative DNS system that provides improved DNS reliability while allowing customizable security policies. These policies can be tailored by administrators, on a per-domain basis, allowing for very strict security at important sites (e.g. banks and online payment sites), while allowing lower-overhead policies for casual browsing. We also make the following contributions:

- We analyze a running cooperative global DNS system, CoDNS, to understand what kind of traffic these resolvers experience. We find that DNS traffic changed qualitatively in the recent past, and that cooperative DNS systems exhibit traffic patterns unlike that reported for individual DNS resolvers.
- We continuously monitor domains around the world from multiple vantage points, allowing us to reverse-engineer routing decisions for their content distribution networks (CDNs) and data centers. We also observe how the DNS mappings used by these domains change over time.
- We present a range of security policies for ConfDNS, show what fraction of domains can be handled by each policy, and show the performance and traffic overhead of each policy.

Even in its weakest configuration, ConfDNS provides better security than local DNS for 99.8% of queries, while stronger security requirements can be met in over 99.2% of queries. On a practical level, ConfDNS is incrementally deployable, and requires no change to the existing global DNS infrastructure to reap its benefits. At the same time, it also provides benefits even if the DNS infrastructure changes to support authentication, using schemes such as DNSSEC [6, 4, 5].

The rest of this paper is organized as follows: in Section 2, we describe ConfDNS, describe how it operates, and what kinds of protection it provides. We then describe

the workload observed on an existing cooperative DNS system, CoDNS, in Section 3, and the studies of the DNS hierarchy performed using the CoDNS workload in Section 4. We discuss our implementation in Section 5, and evaluate some sample ConfiDNS policies in Section 6. We then discuss related work in Section 7 and conclude.

2 Overview

Before describing the workings of ConfiDNS, we describe the Domain Name System (DNS) and the terminology we use for its components. DNS maps human-readable machine names to numeric IP addresses using a globally distributed hierarchy of servers, each of which is responsible for a portion of the global namespace. This system, which we call the server-side DNS infrastructure, is operated by the owners of domain names (e.g. example.com) and their surrogates, and by organizations that are responsible for the top-level servers (e.g. com) that point to the per-domain servers. Clients rarely query these machines directly, but instead send DNS lookups to machines within their own organization, called local (client-side) DNS resolvers. These resolvers perform the queries and cache the results, sharing lookup overhead across many clients. The CoDNS system observed that many DNS problems were due to failures at the local DNS resolvers. CoDNS achieves better performance and reliability by brokering queries to peer DNS resolvers at remote sites when the local resolvers are failing, since resolver failures at different sites were largely uncorrelated.

ConfiDNS attempts to increase confidence in DNS lookups by using peer sites at *all* times in order to provide protection against certain attacks and failures. Additionally, ConfiDNS also uses lookup history to detect changes in name-to-IP mappings. The basic idea is simple – users run a ConfiDNS agent which contains configurable DNS lookup policies. This agent is ideally run on the user's own machine, but can be run on a (possibly shared) machine near the user (with some increased risk). This agent receives DNS lookup requests from the user, and sends the request to both the local DNS resolver as well as some number of peer ConfiDNS agents located at remote sites.

Using the response from the local DNS resolver, the peer ConfiDNS agents, and the agent's recorded history of previous lookups for the name, the agent provides the client with a response, or indicates a failure if no response could be provided that met the specified security policy. Response lookups are also logged to determine lookup history. Sample policies for ConfiDNS include the following: (a) the local resolver and at least one peer must agree on the result, (b) at least three sites must agree, (c) if no peers agree with the local resolver, the IP address must not have changed in the past week, (d) if no peers agree within 5 seconds, use any result. Some questions that naturally arise are

1. What attacks or failures do these policies handle?
2. What sites are amenable to various policies, and what kinds of overheads are incurred in using ConfiDNS?
3. How do these policies interact with content distribution networks or load-balancing schemes, which route traffic using active DNS-to-IP mappings?

2.1 Threat Model & Attacks Handled

ConfiDNS is designed to protect against attacks or failures at the client-side DNS infrastructure, including forms of cache poisoning, compromise, non-failstop failure, and spoofing [16]. These failure modes are real – as recently as several months ago, a new spoofing attack was discovered against the most recent BIND. By protecting the client-side DNS infrastructure, ConfiDNS reduces avenues for polluting global lookups in cooperative DNS systems. While ConfiDNS is not designed to detect server-side DNS spoofing, if the spoofing is selective or intermittent, ConfiDNS may still provide some protection.

The decision to focus on client-side problems is pragmatic – we believe that client-side problems are harder to manually detect, and easier to automatically defend. For example, if an attacker compromises a bank's DNS servers and redirects all traffic to a spoofed Web site, the bank's Web site will see a sharp and easily-detectable drop in activity. However, an attacker who wants to draw less attention could compromise an ISP's resolver, and redirect only lookups for one bank – the resulting drop in traffic may go unnoticed. We have seen several client-side resolver behaviors that could pollute a cooperative DNS service. In one scenario, we saw a site administrator pollute CoDNS by configuring a resolver to reply instantly to all requests with the IP address of a local webserver that served a page saying that the resolver was being replaced. Unfortunately, if the browser expected an image and received this error message, the web page displayed broken image icons, causing problems. We also measured three other instances of pollution, which are further described in Section 4.3. In all of these cases, the results were returned quickly, so any peer using the resolvers at these sites could find its own lookups poisoned in the process.

ConfiDNS's protection does not extend to arbitrary collusion among peers in the system. We present different policies that show how many peers need to agree on a result before ConfiDNS accepts it, and these policies are designed to tolerate different numbers of failing DNS resolvers. However, if an attacker controls all of the resolvers, or even the local ConfiDNS agent itself, we cannot determine the validity of IP addresses.

Although some techniques for strengthening DNS security have been proposed or deployed, they do not solve all of the problems mentioned above. For example, DNSSEC, which can prevent DNS spoofing by authentication, does not provide any support for distributing service (and distributing authentication), leaving the

infrastructure with the same reliability problems as the legacy DNS systems. Adding Byzantine fault tolerance to DNSSEC can further strengthen security while helping reliability, but requires much more dramatic changes to the infrastructure, and is unlikely to occur in the near future given the slow adoption of DNSSEC to date.

2.2 Applicability

ConfidDNS's ability to provide protection depends on how DNS is used (and abused). While we take a pragmatic view and try to accommodate the greatest range of DNS use, the extreme ends of the spectrum are worth considering, because they also provide interesting perspectives. DNS was originally designed so that the same name would resolve to the same IP address everywhere, and that this behavior could be exploited to provide distributed caching. In this model, ConfidDNS's agreement mechanisms are trivially satisfiable, since all sites should get the same answers when performing DNS lookups. The number of faults that ConfidDNS can handle is therefore limited by the number of ConfidDNS peers. The only question for ConfidDNS, then, is how stable the name-to-IP mappings are over time, since a site can seamlessly migrate servers by running both machines at different IP addresses and then waiting until old cached translations expire.

However, since the mid-1990's, intelligent DNS redirectors have been used for geographic load balancing, such as redirecting clients to nearby data centers [14, 23]. Content distribution networks such as Akamai [3] use very short DNS response TTLs to more aggressively balance load and locality. The number of possible IPs per domain name can number in the hundreds in these systems, since they will try to place servers at most large ISPs. These systems are potentially more problematic for ConfidDNS, since peers at separate ISPs may not agree on any domain names served by CDNs. In the extreme, if all sites used CDNs with servers at every large ISP, ConfidDNS would become much less useful. Realistically, though, we believe that the Web will continue to have a mix of sites hosted at single locations, a small number of data centers, and commercial large-scale CDNs. Our measurements later in this paper gives some indication of the breakdowns among the different approaches.

In this paper, we take a pragmatic approach to designing and evaluating ConfidDNS and assume that while some sites will adopt one style of delivery over another, the basic options available will remain qualitatively similar going forward. For each approach, we will try to provide the most protection possible, while realizing that some uses of DNS will simply be more amenable to our style of protection than others. Our interest is in determining what is the best we can possibly do given an imperfect situation, rather than trying to fight the same uphill battle of trying to replace the infrastructure that DNSSEC and others have tried. In the process, we hope to also gain insight into

how DNS is actually used in practice, since the benefits of ConfidDNS will depend on actual DNS usage patterns, rather than on extreme models of possible usage.

3 Trace Analysis

To study how DNS is actually used and what sorts of workloads need to be considered for ConfidDNS, we perform an analysis of the operating CoDNS system, both in terms of how the DNS hierarchy is being used and what implications it has for caching strategies.

3.1 Building a Global DNS Trace

The CoDNS cooperative DNS system has been operational on PlanetLab [20] since October 2003, and has grown from handling 2 million requests per day on 95 nodes to now handling 5-7 million requests per day across PlanetLab (generally 430-450 live nodes). Its most heavy use comes from the CoDeeN [26] content distribution network, which handles over 25 million requests per day from over 50,000 daily users, but is also used by other PlanetLab researchers as well as members of the public who have installed the CoDNS agent on their personal machines.

To produce a global CoDNS trace, we combine log files from all CoDNS nodes over a 34 day period, producing a log with 1.05 million unique successfully queried names. To mimic a global cache, we reduce this log file by combining multiple lookups of the same domain name on each day. From this global cache, we then create a representative daily log by associating each name with a probability determined by how many days it appears in the global log. So, a name that appears every day in the global log will always be included in the daily log, whereas a name that only appears 17 out of 34 days in the global log would only have a 50% chance of appearing in the daily log. Using this methodology, the daily log is reduced to 132K entries, and is used in place of any particular day's log in the remainder of this paper. Creating the daily log in this manner allows us to build a representative day's traffic for a combined global DNS resolver. It also allows us to understand the limits of how much agreement is possible in ConfidDNS without being bound by *a priori* constraints on how many nodes can resolve each name.

This process intentionally creates a daily log where domain names appear only once. Although a local DNS resolver would see the same requests from multiple clients, we are interested in understanding the cacheability of names, and monitoring them over long periods, so we generate traffic patterns within the day as we desire. This behavior is also more like second-level caches rather than local DNS resolvers, since first-level on-box DNS caches are already commonly used for services like CDNs, mail transfer agents, etc. The on-box caches reduce the load

on the site's DNS resolvers, and isolate any performance impact from these high-demand services.

3.2 Domain Frequency Analysis

We can analyze the frequency and usage of domain names to understand how DNS domains and sub-domains are being used by their owners. Aggregating information across CoDNS is useful for several purposes – (1) it provides a snapshot of the state of the DNS system, (2) it gives some insight into global DNS behavior, whereas most previous studies examined only one or two client sites, and (3) it provides some guidance for designing caching strategies.

Note, however, that these statistics are only for the names themselves, and not for the amount of traffic sent to the site. CoDeeN's traffic is generally similar to the Alexa Top 100 list, but the name lookups and their statistics are more important for DNS caches that are interested in only the names. To give an example, google.com is a very popular site by traffic, but it requires very little space in a DNS cache since it has relatively few subdomains.

From these logs, we can see that domain name popularity, measured by how many days a name appears in the trace, is very bimodal. Figure 1 shows counts of how many names appear for a specified number of days in the 34-day trace, while Figure 2 gives the same counts only for the subset of names that appear in the daily trace. In the 34-day trace, most names (53.5%) appear exactly once, while a small set (12.9%) appear on at least 75% of the days, and a very small fraction (1.4%) appear every day. The trend is similar for the daily trace – 14334 names from the one-day trace appear every day in the 34-day trace, and 16559 names in the one-day trace appear only once each in the 34-day trace. The strong bimodality from the 34-day trace is diluted, since pruning reduces the relative weight of the singletons by a factor of 34.

The implications of this distribution are that aggressive caching of DNS names, particularly for global DNS resolvers, may be wasteful. As we see later, most name translations have relatively short TTL values, so caching infrequently-used names provides no benefit. The traffic to cache and find them, only to discover the mapping is stale, may exceed the traffic to simply re-fetch them from the server-side DNS infrastructure.

To understand why relatively recent studies of DNS traffic did not reveal this kind of bimodal distribution, and actually indicated a Zipf-like behavior [12], we examine the most-requested and most-populated domains, shown in Table 1. The counts indicate how many unique names appear under each domain, and the list shows many vanity sites (web logs) and hosting sites, often used to upload images and videos. Most of these sites are younger than the previous DNS studies (performed in 2001), and use DNS very differently, placing the user's name in the DNS name itself rather than as a subcomponent of the URL. This change causes a change in DNS name usage

as millions of people create vanity sites. Since CoDNS is globally deployed, these logs also capture vanity sites in France (free.fr), Russia (narod.ru), and the Arab world (jeeran.com), something not captured in previous studies that only study traffic from one or two sites. The appearance of portals such as Yahoo and QQ (a Chinese portal) on these lists is because they expose cluster names via DNS – these names may map to actual machines, or may just be used for load balancing, but in either case, this partitioning is DNS-visible.

3.3 Caching Implications

We mine this data for implications on caching, since the two existing global DNS systems, CoDNS and CoDoNS, take very different approaches to caching. CoDNS relies on the caches of local DNS resolvers, and performs no caching itself, since it found that paging delays in the virtual memory system were one of BIND's [11] biggest performance and reliability problems. [19] CoDoNS, in contrast, aggressively caches lookup data on each node, in order to reduce the amount of communication needed to return lookup responses.

We query the origin DNS servers to obtain the TTL value for each domain name in the one-day trace, and find that the most common values cluster around 5 minutes, 30 minutes, 1 hour, 4 hours, 8 hours, 12 hours, and 24 hours. We show the breakdown of names in the one-day trace by their TTL groupings in Figure 3. We further select the 2.6% of names that appear every day in the 34-day trace and show their breakdown in Figure 4. In both cases, we add an extra category for those names that had a TTL greater than 24 hours. We observe three important features in this data:

1. Very few translations are cacheable for more than one day, so one day's storage will avoid almost all capacity misses – even if each record requires 64 bytes, all 132K entries in the one-day trace require less than 9MB per node. In practice, nodes will require even less memory, since the one-day trace represents a *combined* trace from all nodes.
2. Most TTLs are four hours or less, so the most popular names will be re-fetched at least six times per day. Predictively refreshing these entries may avoid lookup delays.
3. Many names have TTLs near five minutes, suggesting the use of commercial content distribution networks. The TTL breakdown for the names that appear daily show a greater fraction having a 5-minute TTL than names in the one-day trace. This shift is not surprising, since the most popular sites are prime customers for CDNs.

4 Continuous Monitoring

To obtain a more comprehensive view of how the global DNS hierarchy is used, we perform lookups from multiple vantage points. From this data, we can then analyze

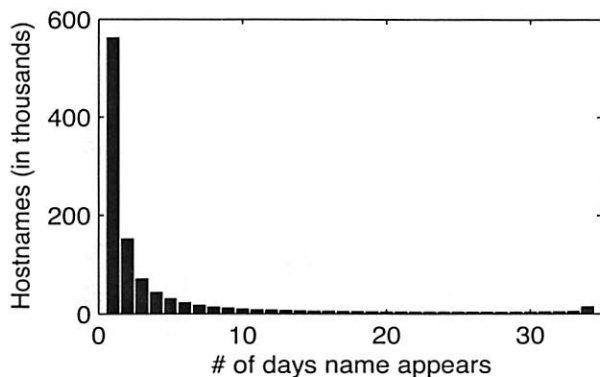


Figure 1: The number of days that each hostname appears in our 34-day survey of CoDNS.

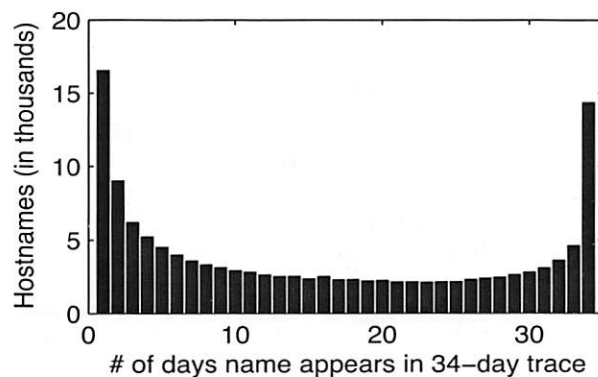


Figure 2: How often hostnames in our one-day trace appear in the monthly trace

34-day (1.05M names)				Daily (132K names)			
top-level		domain		top-level		domain	
592032	com	6642	blogspot.com (V)	82588	com	880	llnwd.net (H)
97948	net	4225	free.fr (H)	12909	net	854	yahoo.com (P)
49343	de	3898	camarades.com (V)	4810	de	494	imageshack.us (H)
44367	org	3775	jeeran.com (V)	4265	org	418	camarades.com (V)
40271	ru	3623	spylog.com (T)	4109	ru	390	blogspot.com (V)
19273	uk	3116	yahoo.com (P)	1848	uk	329	spylog.com (T)
17758	cn	2621	narod.ru (H)	1808	jp	296	free.fr (H)
16750	info	2351	infoseek.co.jp (P/V)	1780	cn	223	jeeran.com (V)
14847	jp	2255	tripod.com (H)	1458	info	220	2o7.net (T)
12054	nl	1413	fastturning.com (T)	1194	nl	200	qq.com (P)

Table 1: Statistics on the most-requested (from the daily trace) and most-populated (from the 34-day trace) top-level domains and regular domains. The count near each name indicates how often it occurs in the set. The indicators near each name indicate the type of site, with V for vanity/weblog, H for hosting, T for tracking, and P for portal. The unique names in the 34-day trace are not always related to aggregated request frequency, as in the case of fastturning.com, which now appears to be non-operational. Also, since this data was gathered, camarades.com has been purchased by another site.

how different usages of DNS affect potential ConfIDNS policies. This approach can give us more insight into the DNS operations of CDNs, as well as the rate of change of name-to-IP mappings.

4.1 Monitoring Setup

To obtain a more comprehensive view, we perform lookups in many distinct locations by using every site in PlanetLab that has its own local DNS resolver. A PlanetLab site consists of two or more co-located nodes; we refer to sites instead of nodes because our choice of node is made dynamically, dependent on which nodes are alive at each site during the trace initialization. After removing dead sites and sites that share DNS resolvers, we are left with approximately 180 accessible sites on most days. Over a period of 30 days, we have each site perform lookups from the one-day trace we described earlier.

To reduce the chances of causing problems for PlanetLab sites, we take a number of steps to make this process

more manageable. To reduce local DNS resolver load, we query only one new hostname per second, and allow only 10 outstanding requests. At that rate, resolving 132K names would require over 36 hours, so we randomly select a 40K name subset, but ensure that all of the Alexa Top 100 are included. With this reduced list, we expect to perform all lookups in less than 12 hours at most sites, and no more than 24 hours at the slowest sites. We take two steps to avoid tripping overly-sensitive intrusion detection systems and overloading DNS servers with synchronized lookups. First, we randomize the list to reduce the chances of querying long bursts of names to the same domain. Second, we have each Planetlab site pick a random starting position in the list. This same starting position is used each day so that the same name is resolved at approximately the same time on a per-site basis.

We performed the monitoring using all available sites for 30 days, and collected information such as the elapsed time per lookup, the IP addresses returned, and the canon-

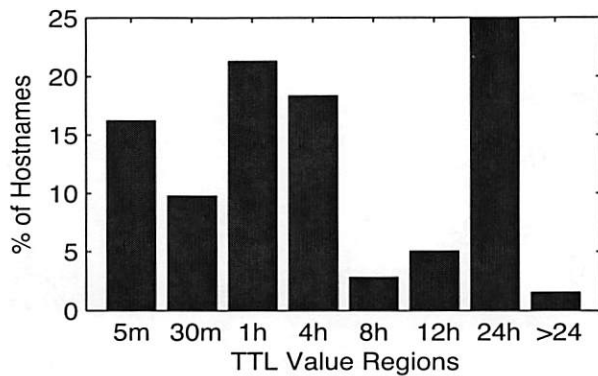


Figure 3: TTL values of hostnames that appear in the one-day trace

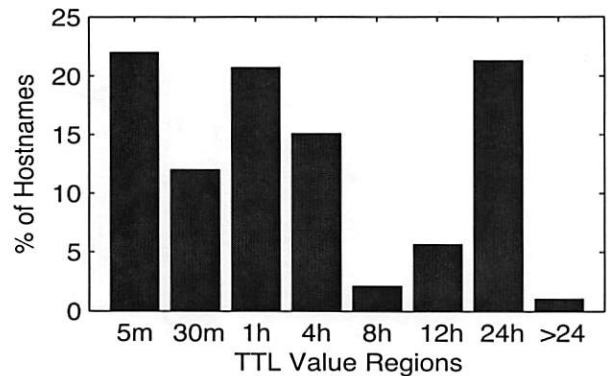


Figure 4: The TTL values of hostnames that appear daily in the 34-day trace

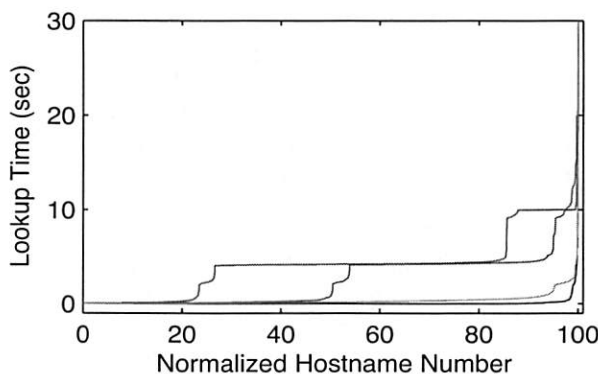


Figure 5: Comparison of Local DNS response times at four sites.

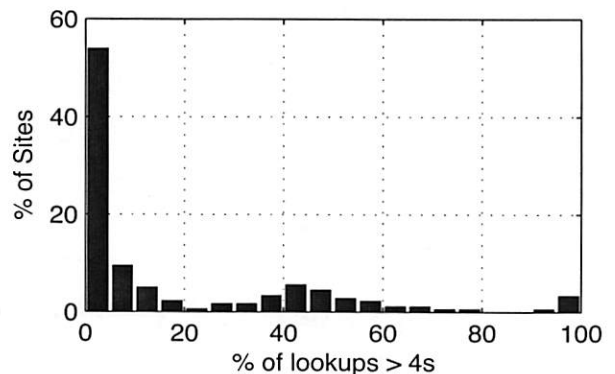


Figure 6: Breakdown of node retry percentages, where DNS lookup times over 4 seconds implies retry.

ical name returned. We also had each node ping its neighbors, and collected round-trip ping times between the PlanetLab nodes. This information is used as the input for the analysis in the rest of this section.

4.2 Query Response Time Breakdowns

These tests give some insight into what range of DNS response times most clients experience when using their local DNS resolver. This measure is important because it provides a baseline comparison for the policies implemented in ConfiDNS. Figure 5 shows the query response times for our set of names at just a few sites, with the values sorted by time to illustrate some common query response behaviors.

We observe three plateaus near zero, five, and ten seconds, and a sharp rise near the end of each line. The plateaus are the timeout values used by the resolver libraries to reissue DNS requests. The plateau shape is due to the fact that most lookups require less than one second when successful, so the response time can be dominated by the retry delay. A larger plateau near zero seconds is desirable, because it indicates that the resolver is either

caching very well, and/or is having no problems in the paths between it and the server-side DNS infrastructure.

To obtain a broader perspective of DNS query performance, we can reduce the results from each site to a single statistic – what fraction of sites require more than a given amount of time to complete their query. We choose 4 seconds as the threshold because it conservatively separates retries from non-retries. We calculate the breakdown for each node, and show a histogram of node failure rates in Figure 6. While most sites (almost 100 out of 180) perform reasonably well, a surprisingly large number (approximately 30) have problems with nearly half of their queries. An additional 7 sites appear to have primary resolvers that are almost entirely non-responsive, where more than 90% of queries take longer than 4 seconds.

Notice that this breakdown is worse than the local DNS resolver health measured in the earlier CoDNS work. The explanation for this difference is that CoDNS monitoring measured the failure rate of cached queries only, to separate resolver failures from cache capacity and replacement policies. Once these extra factors are included in

the end-to-end measurements, many local DNS resolvers perform much worse. However, some servers still perform quite well, with failure rates in the low single digit percentages. Given that a sizable fraction of the ConfiDNS queries are for unpopular names, and we can expect these to be evicted from cache regularly, we can see that the DNS server-side infrastructure is not the major source of problems in the system. If it were, few nodes would be able to achieve low retry rates given the fact that our querying process takes 12 hours and that it queries a number of names unlikely to be cached.

4.3 CDNs and Data Centers

The amount of protection-by-agreement that ConfiDNS can provide is a function of how many clients receive the same IP address for each name. For a content provider that has only one server and one connection to the Internet, all clients should receive the same information when resolving its DNS name. However, providers will often use some form of geographic replication, with multiple data centers or a content distribution network. In these cases, the total number of IP addresses that map to the same name can be much larger, especially if the CDN attempts to have a point-of-presence at every major ISP.

The precise number of IP addresses returned per hostname is less important than the pattern of how they are returned. A domain may have two data centers with addresses IP_1 and IP_2 . If it selectively returns one IP address based on load or locality, we say that the hostname has two *regions*, but if it returns both addresses for every query, we say that it has a single region because all queries see the same set of addresses. At this level of analysis, a single data center with two connections to the Internet (commonly called a multi-homed site) is equivalent to a domain with two data centers but which returns both IPs to every query. Of the roughly 40K domains we query, we find that roughly 90.8% return the same single IP address to all queries. Another 4.2% return multiple IP addresses, but return the same set of IP addresses to all queries.

These statistics are very positive for ConfiDNS because 95% of domain names in the one-day trace can have agreement bounded only by the total number of nodes participating in ConfiDNS. The remaining 5% of domain names (2002 out of 40154) are not automatically out of reach for ConfiDNS – for small numbers of regions, they should be satisfiable for a number of policies.

The exact breakdown of the 2002 hostnames with two or more regions in our one-day trace is shown in Figure 7(a), with hostnames further differentiated into names served by Akamai (354 hostnames) and names either operating on their own or served by other CDNs (1648 hostnames). Akamai-served hostnames are determined using the canonical name returned in the DNS query – any names on akamai.net, akamaiedge.net, speedera.net,

IP Prefix	# sites	Region	Sample Sites
74.125.47.x	2	Southeast US	Georgia Tech
74.125.45.x	3	Kentucky	UKY (Lexington)
74.125.19.x	9	Southwest US	UCSD (San Diego) & ASU (Arizona)
72.14.253.x	2	Wash, Oregon	U Oregon, WSU
72.14.215.x	1	Switzerland	U Zurich
72.14.205.x	4	NYC, Conn	U Conn
66.249.93.x	6	Europe	Vrije U
66.249.91.x	7	Northern Europe	U Helsinki
66.249.89.x	8	Japan, TW	JAIST, Osaka U
66.102.9.x	4	Portugal	U Lisboa
64.233.189.x	5	HK, Korea & China	CUHK (Hong Kong) & Southeast U (Nanjing)
64.233.183.x	9	Western Europe	Cambridge, & UPM (Madrid)
64.233.169.x	25	Eastern US	CMU, Duke
64.233.167.x	35	Great Lakes & Midwest	U Toronto, & Indiana
64.233.161.x	9	Northeast US	NEC Labs (Princeton), & MIT
209.85.193.x	1	Brazil	RNP (Brazil)
209.85.175.x	2	Asia	NTU, SNU
209.85.173.x	15	Western US & Canada	U Washington Intel Res. Berkeley
209.85.135.x	8	Europe	Fraunhofer, LIP6
209.85.137.x	35	Germany, & Austria	U Austria, & TU Darmstadt

Table 2: Breakdown of regions observed for www.google.com with representative sites belonging to each region set

akastream.net, akareal.net, or yimg.com (Yahoo's images served by Akamai) are grouped together as Akamai.

Of the non-Akamai hostnames, 1019 have only two regions, 1429 have ten or fewer regions, and 219 have more than 10 regions. In contrast, most Akamai-served hostnames have 80-90 regions, with Akamai's servers for America Online showing the largest number of regions. Akamai also offers a DNS redirector that resolves to the customers own data centers. These hostnames are handled by akadns.net, and we count these as non-Akamai CDNs since the customer is ultimately handling the actual data centers.

In Table 2 we show a reasonably popular domain - www.google.com, where we find 21 distinct regions visible. The wide range of region sizes implies that different policies will be effective in different parts of the world. Multi-site agreement may work well in the Western United States where there are a sufficient number of available peer sites, but is unlikely to be effective along, for instance, Brazil where there are relatively few available peers.

By comparing the number of sites in each region versus how many would appear in perfectly-balanced regions, we calculate a region imbalance factor for each multi-regioned name in our trace. Given the set of regions with

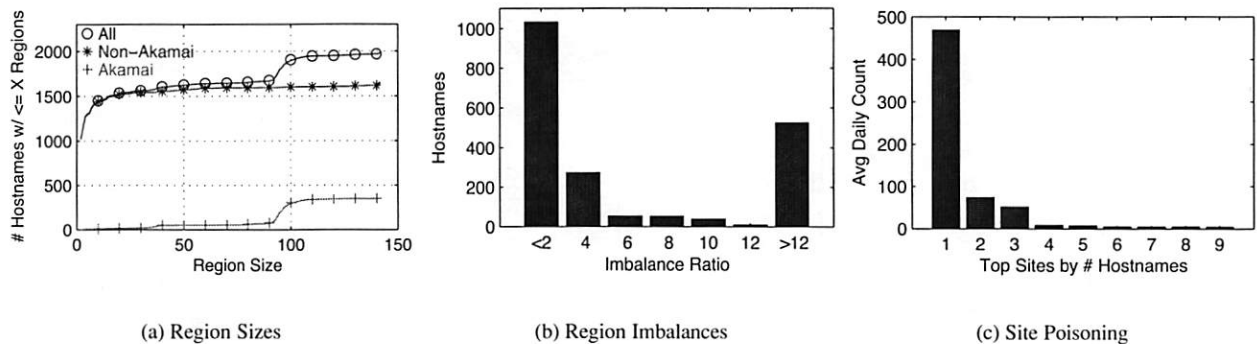


Figure 7: Region information for names mapping to multiple disjoint IP addresses

the number of nodes per region, we calculate a geometric mean of a series of terms, where each term is either the ratio of actual region size to average region size, or its inverse, whichever is larger. For example, if a hostname has three regions with 15, 55, and 80 sites, its imbalance ratio is $\sqrt[3]{\frac{50}{15} \times \frac{55}{50} \times \frac{80}{50}}$. This calculation is designed to identify names that have gross imbalances in the sizes of their regions. While most names have region sizes that are within a factor of 2-4 of being fully balanced, we see a spike where the imbalance ratio exceeds 12 – in these cases, only one site disagrees over the IP address, and all of the other sites form a second region.

To get a sense of the origins of these heavily-imbalanced regions, we counted how often a site disagreed with all others, and show the daily average for the top ten sites in Figure 7(c). The worst site has an average of 469 hostnames per day whose lookups differ from all others. This set of names is fairly stable, and an examination of their contents suggests that it is policy-driven censorship, since they are resolved to IP addresses that provide no responses. Users will be able to seemingly resolve the name, but will be unable to contact any machine at the address, and may conclude the server does not exist. The second-worst site appears to have a traffic-sniffing virus checker working in conjunction with the local DNS resolver. When it activates, all lookups from the client are directed to a local Webserver with a message warning that your client is infected. Unfortunately, the virus sniffer returns false positives, and indicated that our Linux-based boxes were infected with Windows viruses. The third-worst site appeared to be having sporadic failures, and was randomly returning the IP address of the school's main Web server for queries, with no discernible pattern to its behavior. The remaining sites show no strong patterns of poisoning, with most of the imbalances stemming from slowly-deployed changes in name-to-IP mappings. In all of these cases, any multi-site agreement policy in ConfiDNS would automatically prevent these sites from poisoning the lookup results.

4.4 IP Address Changes

By monitoring for 30 days, we gain some insight into how often name-to-IP mappings actually change, rather than just relying on the advertised TTLs as an estimate. While it might be inadvisable to use past history to serve stale mappings, the fact that a mapping has been the same for an extended period of time may give users more confidence in it. Conversely, a mapping that suddenly changes when it had previously been stable may be cause for concern – it may be as simple as a server being replaced or migrated, or it may be that an attacker has spoofed a response and is trying to divert traffic.

From the monitoring data, we calculate the observed rate of change of name-to-IP mappings during our test period. For each site, we examine the results of the lookup for each name across 30 days, and count the number of times the returned IP differs from the previous day's value. For names that map to multiple IPs, we conservatively consider the result to have changed if any member of the set of IPs change. However, we do not consider set ordering important. Since we monitor the mappings only once per day, our approach should be considered a reasonable estimate of the rate of change rather than the precise answer. A domain with a short TTL could presumably change and then revert its mappings between our measurements, and we would miss the change. However, since the probe order differs at each of our measuring sites, such a change would likely trigger our system to detect a change in the number of regions for the domain. If both changes occurred during the 12 hours we do not monitor, the likelihood of our observing it decreases. Given the patterns we observe and our intended uses for the data, we do not believe these corner cases are much of a concern.

The rate of change for all names across all sites is shown in Figure 8(a). For each site, we group names by the number of changes observed, and then report the size of these groups. We see that at every site, more than 85% of names do not change over the 30 day period, despite having TTL values less than 30 days. The remaining bars

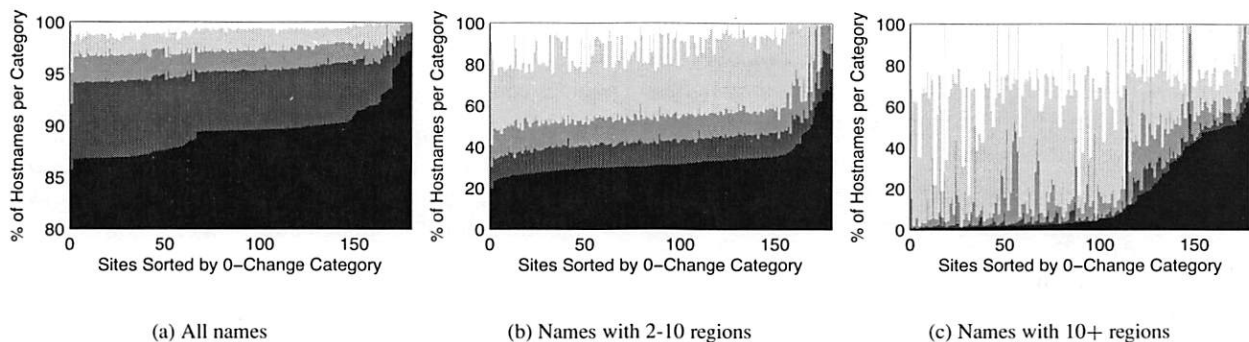


Figure 8: Rate of change for name-to-IP mappings. The bars, from bottom to top, are for zero changes, 1 change, 2-3 changes, 4-14 changes, and 15+ changes (in white). **Note: Y axis truncated to improve detail in first graph.**

group the number of changes and are intended to show that while some names change on virtually every lookup, most change much more slowly. If these rates of change are typical, then we see that most names are stable for a month at a time, and more than half of the names that do change are in fact, stable for two weeks. The set of names that change more frequently than once per week ranges from 1-3% in this study.

The data in Figure 8(b) breaks out the names with a small number of regions (2-10), so that we can determine how often most multi-regioned sites send clients to different regions. We see that even here, a large number of names have long periods of stability – decisions to send clients to nearby data centers are likely to be stable over time, unless the closest data center becomes unavailable for maintenance, link outages, etc. The names that show high rates of change in these measurements may indicate that in addition to (or instead of) geographic proximity, the DNS server is also being used to load balance traffic across multiple data centers.

Finally, the data in Figure 8(c) shows the same statistics, but only for those names that map to more than 10 regions. Included in this set are most of the Akamai-served domains, some domains served by LimeLight Networks (another CDN) as well others that seem to be using a fairly large number of their own data centers (or hosting centers). The increase in the size of the zero change category beginning near node 100 is largely a function of the size and deployment of Akamai clusters – these do not appear to use hardware load balancers, so the larger the cluster, the more IP addresses that are exposed and rotated, causing high rates of IP address changes. In contrast, Google clusters, despite having thousands of nodes, advertise only a small number of IP addresses as entry points.

As can be seen from this data, IP result history can be used profitably to provide an indication of stability, and this pattern holds true across domains served using a variety of different strategies. This observation bodes well

particularly for those parts of the world where sparse coverage may preclude certain clients from finding enough peers in agreement to use only agreement-based policies. In these cases, the stability of a name-to-IP mapping can provide some reassurance greater than just the local DNS resolver alone provides.

The stability data we have gathered may also be useful in shaping decisions about when to use stale DNS data, as has been proposed in another system [10]. If the DNS infrastructure and the actual content servers do not share fate, it may be the case that a domain has functioning servers but is not accessible to users because their existing DNS entries have become stale. Note that this fate-sharing requirement is not completely unrealistic – several companies perform outsourced DNS service, so a failure at a third-party DNS provider could have no correlation to the domain's own content servers failing. In these cases, a DNS resolver could potentially be configured to provide stale data if it met some predefined criteria. We do not explore this idea further in this paper, but leave it as a possible avenue for future work.

5 Implementation

ConfiDNS is implemented as a service running on PlanetLab, with an architecture similar to the CoDNS system. Each PlanetLab node runs a ConfiDNS agent, which can also be run on the user's local machine. The agents accept DNS queries using TCP and UDP, and when run on a local machine, can automatically modify the `/etc/resolv.conf` file to automatically handle all locally-generated DNS traffic.

The policy differences between CoDNS and ConfiDNS focus on when to contact peers. In CoDNS, requests are first forwarded to the local DNS infrastructure, and only sent remotely if the local resolver does not respond within an adaptive time-out period. The exception to this policy is if the local resolver is deemed dead, in which case all requests are immediately forwarded to a peer. In ei-

ther case, CoDNS contacts successive peers only when the previous peers fail to respond in an exponentially-increasing timeout period. Peer selection in CoDNS is performed from a set of nearby nodes using the Highest Random Weight (HRW) hashing scheme [25][19], in order to preserve cache locality.

In ConfiDNS, all locally-generated queries are immediately forwarded to a specified number of peers, chosen purely based on proximity using a heartbeat ping. The decision to use proximity alone was motivated by the hope that peers that are closer are more likely to be in the same CDN region. Additional queries are sent only when the agreement policy has not been met within a give timeout. As with CoDNS, queries that are received from remote nodes are not re-forwarded, both to prevent forwarding loops, and to limit the damage that any compromised peer can cause.

In keeping with the desire to prevent any pollution from spreading within ConfiDNS, only locally-resolved lookup results are stored for keeping history and satisfying query requests. Both locally-generated and remote queries can be satisfied from the cache, but only locally-generated results are entered into the cache to avoid pollution.

ConfiDNS uses a configuration file that specifies domain name suffixes and policies, so that policies can be customized as needed. Possible policies include the first response, agreement of N out of M peers, and historical agreement. Multiple policy lines can be provided with different start times, so that one can opt for different decisions if a previous policy is taking too long to satisfy. Canonical names and IP addresses can also be specified, allowing the whitelisting of any Akamai-served name, or just Akamai-served names from a given set of IP addresses. ConfiDNS is responsible for determining when the specified lookup policy has been satisfied. If no agreement is reached between the set of remote peers, the ConfiDNS agent sends a failure response to the client, but does not cache the result. We have thought of having the failure response direct the client to a locally-configured Web server that can explain why the lookup failed (using out-of-band information), but have not implemented this approach. The benefit of this scheme is that the user would then be able to see why a given policy could not be satisfied, and could then choose a different action, including possibly choosing a new policy or reporting any anomaly to a system administrator.

6 Evaluation

In this section, we evaluate a number of ConfiDNS policies, first examining policies that relate only to agreement, and then combining agreement and history. Our primary focus in this evaluation will be coverage (applicability) and latency – we choose policies that are designed to have reasonable network overheads, so our initial analysis will focus on how many domains and how many sites bene-

fit from the various types of security the different policies provide. Given the observations in CoDNS about trading latency for network overhead, we believe that all of these policies can be tuned as required.

We evaluate four agreement policies for ConfiDNS, requiring agreement among varying numbers of peers from various maximum peer-set sizes. In one sample policy, we require that the set of agreeing peers include the local DNS resolver. In the rest, the local DNS is just one of several peers that may participate in the agreement process. Each peer consists of a single node at any particular PlanetLab site that possesses a locally managed (non-shared) DNS resolver. In each case, we place a restriction on the number of peers that can be queried in order to reach agreement. Peers are ranked using their round-trip times, and we choose the set of peer nodes with the lowest RTTs. We use this ranking method to choose our peer sites both for the obvious reason, to minimize query response time, but also to reduce spurious results being returned for multi-region address mappings. For example, if we require five peer agreement on a response to a DNS query without a locality restriction on peering, nearly every domain name can meet the agreement requirement so long as we choose a reasonably well localized set of peers. These peers are not necessarily within our own region, indeed they may be anywhere in the globe. Directing all traffic to that potentially far flung region is unlikely to be a desirable property, both from the point of view of the end-user, who may see a degradation in service performance at a particular host, and also from the point of view of a service operator, who will find DNS-based load-balancing to be less effective with potentially many users circumventing the redirection.

All policies are evaluated at every site, and per-site average latencies are reported in Figures 9– 14. The baseline policy, using only the local DNS resolver, suffers from the problem of retries that we described in Section 4.2. Likewise, the policy of requiring that at least one other site (out of the five closest peers) agree with the local resolver shows similar performance because the local DNS response time is the bottleneck. A simplified form of the CoDNS policy is shown in Figure 11, and takes the first response of the local resolver and the three nearest peers. The dispatch of queries to the peer sites are staggered using the same delay values that CoDNS uses in deployment. This policy (as does CoDNS) shows a significant response time improvement over the local DNS resolver, precisely because we do not have to wait for the local resolver's response when it is slow.

The more aggressive agreement policies for ConfiDNS require 3 of the 10 closest sites agreeing (Figure 12), 5 sites out of 20 agreeing (Figure 13), or 7 sites out of 30 agreeing (Figure 14). Though we are primarily concerned with the level of agreement possible, we stagger the dis-

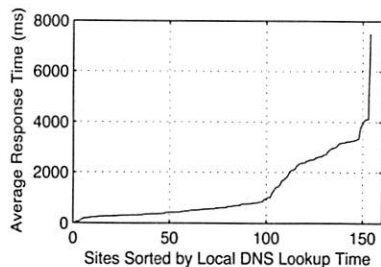


Figure 9: Local DNS resolver only.
Note 8000ms Y axis

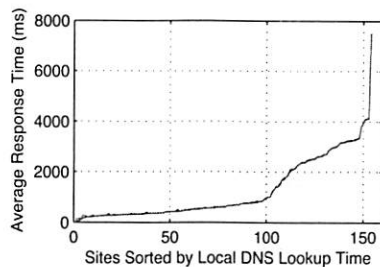


Figure 10: Local + 1 site from 5 peers. Note 8000ms Y axis

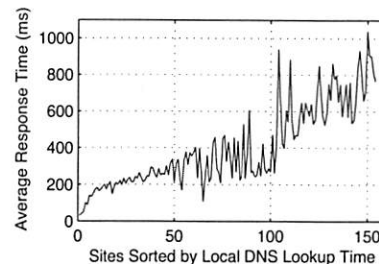


Figure 11: CoDNS

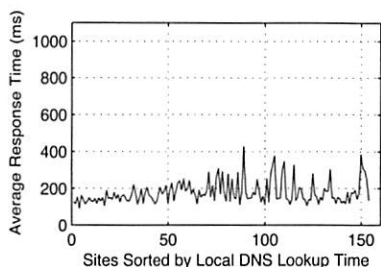


Figure 12: 3 sites from 10 peers

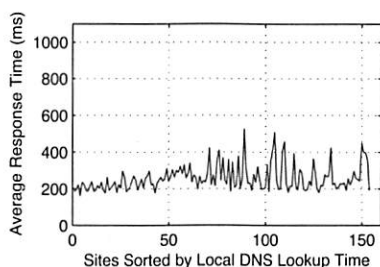


Figure 13: 5 sites from 20 peers

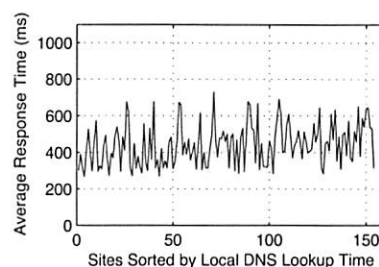


Figure 14: 7 sites from 30 peers

patch of queries in these policies at the rate of 1000ms every 10 lookups for fairness. A practical, low-overhead policy of taking either the local DNS resolver and one peer, or 3 other peers, is not shown because its latency characteristics are identical to Figure 12.

To evaluate our sample implementation of ConfiDNS we once again sampled traffic from CoDNS, and replayed a previous day of CoDNS lookups using the same request timing as in the original trace, but with the previous days names reduced to a single unique lookup per name. This resulted in approximately 20,000 unique names resolved at a frequency determined by the original pattern of traffic. Figure 15 shows the per-site average response time, compared to the same trace resolved at the same sites using CoDNS. We can see that the actual performance is similar to that predicted from our trace-based analysis.

The most important latency observation for these more secure policies is that they perform much better than local DNS resolvers, and are in fact generally better than CoDNS. The 3-agreement policy performs surprisingly well compared to CoDNS, with an average latency almost half of CoDNS's. This is partly due to the fact that CoDNS initially waits 200ms before dispatching peer queries (on nodes with healthy local resolvers) while ConfiDNS dispatches these queries immediately. The 5-agreement policy performs roughly 25% worse than 3-agreement across all sites, but is still better than CoDNS. The 7-agreement policy performs another 60% worse in general, but is otherwise comparable in latency to CoDNS. These results show that the ConfiDNS policies can produce good laten-

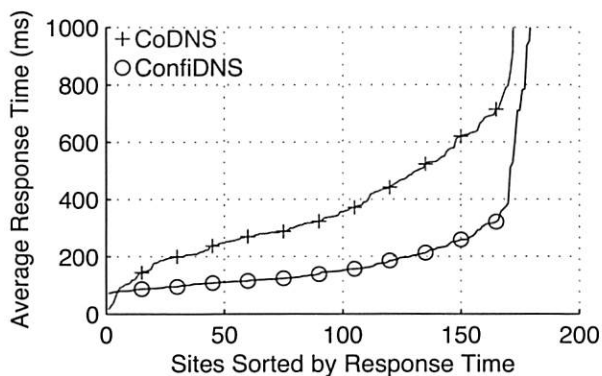


Figure 15: ConfiDNS implementation 3 sites from 10 peers vs. CoDNS

cies, even for relatively high levels of agreement. If the accompanying network overheads are unacceptable, then the queries to the peers can be staggered even more so that network overhead is reduced at the cost of some latency.

Since ConfiDNS query latency is clearly improved over non-cooperative schemes, the other concern is what fraction of domain names can be satisfied using each of the policies, where satisfiability refers to the ability of a particular client to resolve a name with the given policy setting. Since the agreement policies are tied to how multi-region domains behave, the satisfiability concerns are related to the location of the site, the distance to nearby peers, and the granularity of CDN redirection processes

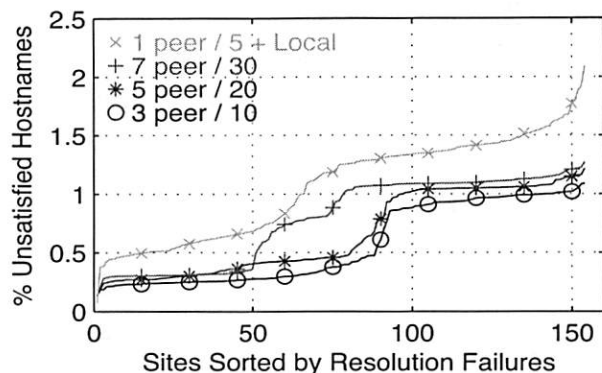


Figure 16: Percentage of names that cannot be satisfied at each site given the particular agreement policy. Data is sorted on a per-policy basis for ease of viewing.

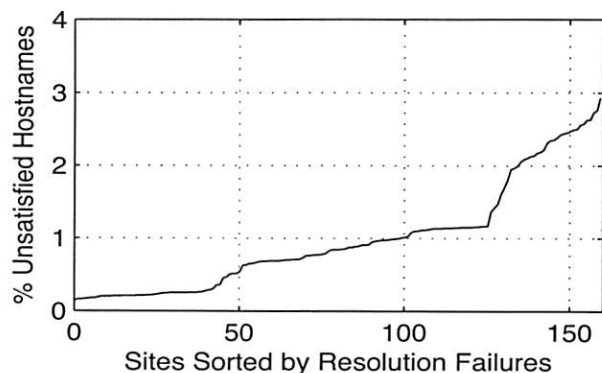


Figure 17: Percentage of names that cannot be satisfied at each site by our ConfiDNS implementation given a 3 peer agreement policy.

in the local area. Rather than reducing these numbers to averages, we present the percentage of domain names that each policy fails to satisfy at each site. This data is shown in Figure 16 for the trace-based analysis, and in Figure 17 for the live implementation on a 3-agreement policy using 10 peers. Again, the live implementation agrees closely with our trace-based analysis.

In general, we see that only about 1% of the domains fail to be satisfied under the various agreement policies, and at one-third of the sites, the rate is even lower, in the range of 0.3%. These results are in-line with what we had observed about the number of regions used by names that had more than one region. Despite multi-regioned names accounting for 5% of the trace, the fact that most of these names have fewer than 10 regions makes them amenable to our agreement policies. The names that are not amenable are the cause of the unsatisfiability rates observed. The plateaus in these graphs are worth mentioning – the lower plateau occurs because of CDN nodes out-

	# Days IP is Stable				
	none	2-3	7	15	30
local DNS only	1	2	3	3	4
3 peers	2	3	3	4	4
local DNS + 1	3	3	4	4	5
5 peers	3	4	4	5	5
7 peers	4	4	5	5	6

Table 3: For ease of analysis, we linearize and collapse the range of protection policies in to single value labels, as shown above. Higher numbers indicate better protection than lower numbers.

side of the United States and Europe. In these areas, the smaller ISPs may not have enough bandwidth for CDN companies to place nodes inside their networks. As a result, it appears that the CDN nodes are in regional networks, and are used by many ISPs, leading to higher agreement rates than within the US and Europe. The other interesting result worth discussing is why the weaker policy of “local DNS + 1 peer” performs poorly – it dispatches queries to only 5 peers, whereas the 3-agreement tries 10 peers, thus creating greater potential for successful agreement. Additionally, forcing the peers to agree with the local DNS is more restrictive.

6.1 Putting It All Together

We can now combine the policy agreement data and the rate-of-change monitoring data to determine the spectrum of protection policies that are possible, and how many hostnames can be satisfied with each. Rather than try to combine the aggregated data we have gathered, we perform the analysis for each hostname on each domain. We have two dimensions, agreement and history, so to simplify the analysis, we linearize the range of possibilities, as shown in Table 3. The process of assigning values to policies is subjective, but our main goal was to give an idea of the strength of combinations, with higher numbers indicating better protection. To get a sense of how often the policies are satisfied, the per-site breakdown for each label is shown in Figure 18.

The average breakdown of labels across sites is shown in Table 4. We can see that the percentage of hostnames that can only be satisfied by label 1, the weakest security policy, averages 0.18%. As label 1 is equivalent to local resolver lookup with no query history, this result indicates that ConfiDNS improves the security of 99.82% of queries. Even if we pick a stronger security requirement, such as label 4, which indicates seven peers agreeing, or 30 days of stability, or some intermediate combinations, ConfiDNS is able to satisfy 99.64% (the sum of labels 4 - 6) of the queries. Even the strongest policy, with 7 peers agreeing and the hostname resolution being stable for 30

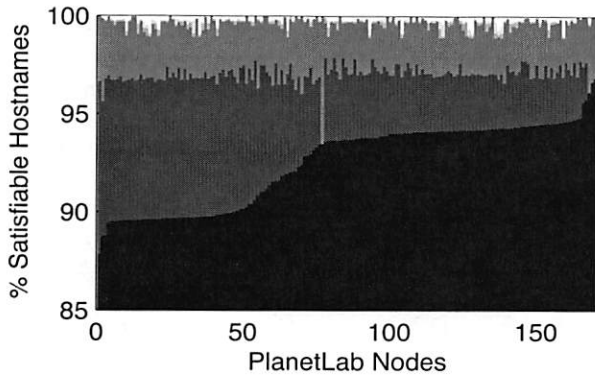


Figure 18: Breakdown of linearized policy labels by site. The bottom bar is linearized label 6, and the top bar (white) is label 1. **Note: Y axis truncated to show detail**

days, still works for over 92% of queries. As previously shown, this extra security does not come at a high cost – latency is better than local DNS alone, and is comparable to CoDNS (which has much weaker security than local DNS only).

7 Related Work

The idea of using some form of replication to achieve fault-tolerance is older than the authors of this paper. In the OS community, this approach has recently seen a revival of interest, especially since protecting geographically-distributed systems against Byzantine failures has become a focus of concern. Examples of such include performance-conscious Byzantine fault tolerance for NFS [8], including Byzantine agreement for quorum systems [17], and scaling cooperative services when facing Byzantine and selfish users [2]. These techniques have also been proposed to secure DNS [1, 27], when used in conjunction with DNSSEC.

These approaches provide strong protection, at the cost of relatively high requirements, such as having heterogeneous systems and components. In practice, for widely-deployed services such as DNS, the number of different configurations (including different components) is low enough that Byzantine requirements may be difficult to meet with the current infrastructure [13].

The research community has recently renewed its focus on improving server-side infrastructure. Cox *et al.* investigate the possibility of transforming DNS into a peer-to-peer system [9] using a distributed hash table [24]. This replaces the hierarchical DNS name resolving process with a flat peer-to-peer query style in pursuit of load balancing and robustness. With this design, misconfigurations from administrator mistakes can be eliminated and the traffic bottleneck on the root servers is removed so that load is distributed over the entities joining the system. In CoDoNS, Ramasubramanian *et al.* improve the

Policy label	Mean %	Std Dev
1	0.18	0.17
2	0.07	0.12
3	0.12	0.13
4	2.59	0.54
5	4.49	2.08
6	92.56	2.16

Table 4: Mean percentage of hostnames satisfied by a particular policy label with corresponding Std Dev. Most lookups can be satisfied by one of the stronger policies (4,5,6) instead of the weaker ones (1,2,3).

latency performance of this approach by using proactive replication of DNS records [22]. They exploit the Zipf-like distribution of domain names in web browsing [7] to reduce the replicating overhead while providing $O(1)$ proximity [21]. Our previous work on this subject includes a workshop paper [15] where we sketch the ideas presented here. We expand this work with an in-depth analysis of our global DNS trace including a discussion of name-popularity its caching implications, as well as an investigation of query response-times. Finally, we outline an implementation that is a base for our future work.

8 Conclusion

Cooperative DNS resolvers have proven their utility in improving reliability and performance when compared to local DNS resolvers, but at the cost of weakened security. In this work, we show that by using peer agreement and storing some past history, our new cooperative resolver ConfiDNS, can provide *better* security than both traditional DNS resolvers and previous cooperative approaches, for the majority of domain names. Using a month-long world-wide survey of DNS behavior on a realistic global DNS trace, we are able to determine the applicability of various agreement policies and quantify their effect on latency. This paper also provides some raw data about global DNS behavior that should be useful to the broader research community. In addition to providing some insight into DNS behavior at scale, we also demonstrate that new uses of DNS by the increasingly popular vanity sites is qualitatively changing the patterns of DNS namespace usage. This study also provides us with information on the real usage of DNS mappings at a variety of domains ranging from small, singly-hosted sites to sophisticated, replicated data centers with DNS redirection, and finally to commercial third-party content distribution networks. In all cases, we find that it is possible to leverage scale, history, or both, and provide a much more secure result than local DNS alone.

These benefits are obtained without changing any

server-side DNS infrastructure, and with a tolerable and tunable network overhead. As a result of our design, ConfiDNS is incrementally deployable, requiring only a minimal agent running on either client machines or on client-side resolvers. Performance and network overhead can be improved by adding a small amount of caching to ConfiDNS, and we quantify the cost impact of a reasonable caching scheme. Finally, our approach is compatible with server-side approaches to improving DNS security such as DNSSEC, and together can provide reliability and performance benefits in addition to improved security.

Acknowledgments

We would like to thank our shepherd, David Presotto, and the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF Grants ANI-0335214, CNS-0439842, and CNS-0520053.

References

- [1] S. Ahmed. A scalable Byzantine fault tolerant secure domain name system, 2001. Massachusetts Institute of Technology Technical Report MIT-LCS-TR-849.
- [2] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. Bar fault tolerance for cooperative services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, Brighton, United Kingdom, October 2005.
- [3] Akamai. Content Delivery Network. <http://www.akamai.com/>.
- [4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Dns security introduction and requirements. Request for Comments 4034, March 2005.
- [5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource records for the dns security extensions. Request for Comments 4035, March 2005.
- [6] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol modifications for the dns security extensions. Request for Comments 4033, March 2005.
- [7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM*, New York, NY, March 1999.
- [8] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.
- [9] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS Using Chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.
- [10] H. Ballani and P. Francis. A Simple Approach to DNS DoS Mitigation. In *Proceedings of the 5th ACM Workshop on Hot Topics in Networks (HotNets '06)*, Irvine, CA, November 2006.
- [11] Internet Systems Consortium (ISC). ISC BIND. <http://www.isc.org/>.
- [12] H. B. Jaeyeon Jung, Emil Sit and R. Morris. DNS Performance and the Effectiveness of Caching. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop '01*, San Francisco, California, November 2001.
- [13] B. Knowles. Domain Name Server Comparison: BIND 8 vs. BIND 9 vs. djbdns vs. ??? In *Proceedings of the USENIX LISA Conference 2002 - Technical Program*, Berkeley, CA, November 2002.
- [14] B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proceedings of SIGCOMM Internet Measurement Workshop*, San Francisco, CA, November 2001.
- [15] L. Poole and V. S. Pai. ConfiDNS: Leveraging Scale and History to Improve DNS Security. In *Proceedings of the Third Workshop in Real, Large, Distributed Systems (WORLDS '06)*, Seattle, WA, November 2006.
- [16] A. Lioy, F. Maino, M. Marian, and D. Mazzocchi. Dns security. In *Proceedings of the TERENA Networking Conference*, Lisbon, Portugal, May 2000.
- [17] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, El Paso, TX, May 1997.
- [18] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, August 1988.
- [19] K. Park, V. S. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.
- [20] PlanetLab. An open testbed for developing, deploying and accessing planetary-scale services, September 2002. <http://www.planet-lab.org/>.
- [21] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.
- [22] V. Ramasubramanian and E. G. Sirer. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proceedings of the ACM SIGCOMM Conference*, Portland, OR, August 2004.
- [23] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of dns-based server selection. In *Proceedings of INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, Anchorage, AK, April 2001.
- [24] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, San Diego, California, August 2001.
- [25] D. Thaler and C. Ravishanker. Using Name-based Mappings to Increase Hit Rates. In *IEEE/ACM Transactions on Networking*, volume 6, 1, 1998.
- [26] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2004.
- [27] Z. Yang. Using a byzantine fault tolerant algorithm to provide a secure dns, June 1999. Massachusetts Institute of Technology Masters Thesis.

Large-scale Virtualization in the Emulab Network Testbed

Mike Hibler Robert Ricci Leigh Stoller Jonathon Duerig
Shashi Guruprasad[†] Tim Stack[†] Kirk Webb[†] Jay Lepreau

University of Utah, School of Computing
www.emulab.net www.flux.utah.edu

Abstract

Network emulation is valuable largely because of its ability to study applications running on real hosts and “some-what real” networks. However, conservatively allocating a physical host or network link for each corresponding virtual entity is costly and limits scale. We present a system that can faithfully emulate, on low-end PCs, virtual topologies over an order of magnitude larger than the physical hardware, when running typical classes of distributed applications that have modest resource requirements. This version of Emulab virtualizes hosts, routers, and networks, while retaining near-total application transparency, good performance fidelity, responsiveness suitable for interactive use, high system throughput, and efficient use of resources. Our key design techniques are to use the minimum degree of virtualization that provides transparency to applications, to exploit the hierarchy found in real computer networks, to perform optimistic automated resource allocation, and to use feedback to adaptively allocate resources. The entire system is highly automated, making it easy to use even when scaling to more than a thousand virtual nodes. This paper identifies the many problems posed in building a practical system, and describes the system’s motivation, design, and preliminary evaluation.

1 Introduction

Network experimentation environments that emulate some aspects of the environment—network testbeds—play an important role in the design and validation of distributed systems and networking protocols. In contrast to simulated environments, testbeds like Emulab [28] and PlanetLab [20] provide more realistic testing grounds for developing and experimenting with software. Emulated environments implement virtual network configurations atop real hardware: this means that experimenters can use real operating systems and other software, run their

applications unmodified, and obtain actual (not simulated) performance measures.

A primary challenge for future emulation environments is scale. Because emulated environments are supported by actual hardware, an emulated system that is “larger” than the underlying physical system requires the careful allocation and multiplexing of a testbed’s physical resources. To avoid experimental artifacts, the original Emulab used strictly conservative resource allocation. It mapped virtual network nodes and links one-to-one onto dedicated PCs and switched Ethernet links. We have four motivations for relaxing this constraint, allowing controlled multiplexing of virtual onto physical resources. First, some applications such as peer-to-peer systems or dynamic IP routing algorithms require large topologies or nodes of high degree for evaluation, yet are not resource-hungry. Second, much research and educational use simply does not need perfect performance fidelity, or does not need it on every run. Third, such multiplexing provides more efficient use of shared hardware resources; for example, virtual links rarely use their maximum bandwidth and so waste the underlying physical link. Fourth, it makes small-scale emulation clusters much more useful.

In this paper we present a collection of techniques that allow network emulation environments to virtualize resources such as hosts, routers, and networks in ways that preserve high performance, high fidelity, and near-total transparency to applications. Our primary motivation is scale: i.e., to support larger and more complex emulated environments, and to allow a single testbed to emulate more such environments at the same time. Our techniques allow a testbed to better utilize its physical resources, and allow a testbed to emulate kinds of resources that it may not have (e.g., hosts with very large numbers of network interfaces). One goal of our techniques is to preserve the performance fidelity of emulated resources, but our approach can also be used in cases where users do not require high fidelity, e.g., during early software development, in education, or in many

[†]Currently at Cisco Systems, VMware, and Morgan Stanley, respectively. Work done at the University of Utah.

kinds of reliability studies. Our techniques provide benefits to both testbed operators, who can provide better services with fewer hardware resources, and users, who have improved access to testbeds and their expanded services.

Our goal is that the overall system *scales well* with increasing size of virtual topologies. Scalability is not only about speed and size, but also concerns reliability and ease of use. We are concerned with (i) the time to reliably instantiate an experiment, which affects system throughput and Emulab's interactive usage model; (ii) the number of physical machines and links required for a particular virtual topology; (iii) monitoring, control, and visualization of testbed experiments, both by the user and the system; and (iv) the user's time spent in customizing instrumentation and management infrastructure. In achieving good scalability, however, we require two constraints to be met. One is that the emulation system be, as much as possible, *transparent to applications*. Even if they deal with the network or OS environment in an idiosyncratic manner, we should not require them to be modified, recompiled, relinked, or even run with magic environment variables. Second, we must provide good (not perfect) *performance fidelity*, so that experimenters can trust their results.

To meet our goals, our enhanced testbed multiplexes virtual entities onto the physical infrastructure using four key design techniques. First, it uses the minimum degree of virtualization that will provide sufficient transparency to applications. Second, it exploits the hierarchy found in the logical design of computer networks and in the physical realization of those networks. Our resource allocator relies on implicit hierarchy in the virtual topology to reduce its search space; our IP address assigner infers hierarchy to provide realistic addresses; and our testbed control system exploits the hierarchy between virtual nodes and their physical hosts. Third, our system employs optimistic automated resource allocation. The system or the user makes a "best guess" at the resources required, which are fed into a resource assigner that uses combinatorial optimization. Fourth, our testbed uses feedback to allocate resources adaptively. In training runs and in normal use, system-level and optional application-specific metrics are monitored. The metrics are used to detect overload or underload conditions, and to guide resource re-allocation. Emulab can automatically execute this adaptive process.

This paper makes the following contributions:

- It describes levels of virtualization that are appropriate for this domain, and discusses the design trade-offs.
- It shows how to solve the NP-hard resource assignment problem for networks of thousands of entities, and describes how to support flexible specification

of arbitrary resources.

- It presents a new feedback-directed technique to support virtualization and scaling.
- It outlines a new algorithm for efficiently assigning realistic IP addresses.
- It provides a preliminary experimental evaluation of various aspects of the system.
- The system it describes provides a useful new facility and is proven in production use.

One of the lessons of our work is that, in practice, achieving such a scalable system requires a collection of techniques covering a wide range of issues. The challenge is more broad and difficult than simply virtualizing an OS or virtualizing network links. It includes solving difficult problems in IP address assignment, in allocating node and network resources, in performance feedback, and in scalable internal control systems. We believe that this fact—that a host of problems must be solved to achieve scalable network experimentation in practice—is not widely recognized.

2 Testbed Context

The Emulab software is the management system for a network-rich PC cluster that provides a space- and time-shared public facility for studying networked and distributed systems. One of Emulab's goals is to transparently integrate a variety of different experimental environments. Historically, Emulab has supported three such environments: emulation, simulation, and live-Internet experimentation. This paper focuses on our work to expand it into a fourth environment, virtualized emulation.

An "experiment" is Emulab's central operational entity. An experimenter first submits a network topology specified in an extended *ns* [6] syntax. This virtual topology can include links and LANs, with associated characteristics such as bandwidth, latency, and packet loss. Limiting and shaping the traffic on a link, if requested, is done by interposing "delay nodes" between the endpoints of the link, or by performing traffic shaping on the nodes themselves. Specifications for hardware and software resources can also be included for nodes in the virtual topology.

Once the testbed software parses the specification and stores it in the database, it starts the process of "swapping" to physical resources. Resource allocation is the first step, in which Emulab attempts to map the virtual topology onto the PCs and switches with the three-way goal of meeting all resource requirements, minimizing use of physical resources, and running quickly. In our case the physical resources have a complex physical topology: multiple types of PCs, with each PC connected via four 100 Mbps or 1000 Mbps Ethernet interfaces to switches that are themselves connected with multi-gigabit links.

The testbed software then instantiates the experiment on the selected machines and switches. This can mean configuring nodes and their operating systems, setting up VLANs to emulate links, and creating virtual resources on top of physical ones. Emulab includes a synchronization service as well as a distributed event system through which both the testbed software and users can control and monitor experiments.

We have seven years of statistics on 2000 users, doing 69,000 swapins, allocating 806,000 nodes. An important observation is that people typically use Emulab *interactively*. They swap in an experiment, log in to one or more of their nodes, and spend hours running evaluations, debugging their system, and tweaking parameters, or sometime spend just a few minutes making a single run. When done for the morning, day, or run, they swap out their experiment, releasing the physical resources.

This leads to two points: speed of swapin matters, and people “reuse” experimental configurations many times. These points are important drivers of our goals and design.

3 Minimal Effective Virtualization

Multiplexing logical nodes and networks onto the physical infrastructure is our approach to scaling. *Virtualization* is the technique we use to make the multiplexing transparent. Our fundamental goal for virtual entities is that they behave as much like their real-life counterparts as possible. In the testbed context, there are three important dimensions to that realism: functional equivalence, performance equivalence, and “control equivalence.” By the last, we mean similarity with respect to control by the testbed management system (enabling code reuse) and by the experimenter (enabling knowledge reuse and scripting code reuse). This paper concentrates on the first two dimensions, functional realism, which we call *transparency*, and performance realism.

Our design approach is to find the minimum level of virtualization that provides transparency to *applications* while maintaining high performance. If a virtualization mechanism is transparent to applications, it will also be transparent to experimenters’ control scripts and to their preconceived concepts. We achieve both high performance and transparency by virtualizing using native mechanisms: mechanisms that are close to identical to the base mechanisms.

For virtual nodes, we implement virtualization within the operating system, extending FreeBSD’s jail abstraction, so that unmodified applications see a system call interface that is identical to the base operating system. For virtual links and LANs, we virtualize the network interface and the routing tables. That allows us to provide key aspects of emulated networks using native switch-

supported mechanisms such as broadcast and multicast. These mechanisms give us high—indeed native—performance, while providing near functional equivalence to applications. In our current virtual node implementation we give up resource isolation, but we are saved by our higher-level adaptive approach to resource allocation and detection of overload.

3.1 Virtual Nodes

There are many possible ways to implement some notion of a “virtual node.” The available technologies and the trade-offs are well documented in the literature (e.g., [19]). When choosing the technology for Emulab virtual nodes, we evaluated each against four criteria, two from an application perspective, two from a system-wide perspective:

Application transparency. The extent to which virtual node name spaces (e.g., process, network, filesystem) are isolated from each other. (Can the application run unchanged?)

Application fidelity. The extent to which virtual node resources (e.g., CPU, memory, IO bandwidth) are isolated from each other. (Does the application get the resources it needs to function correctly?)

System capacity. The amount of virtualization overhead. (How many virtual nodes can we host per physical node?)

System flexibility. The level at which virtualization takes place (can we run multiple OSes?) and the degree of portability (can we run on a wide range of hardware?)

3.1.1 Emulab Virtual Nodes

Application transparency is important in the Emulab environment, requiring at least namespace isolation to be present. On the other hand, we anticipated that the initial network applications run inside virtual nodes would have modest CPU and memory requirements, making resource isolation—except for the network, which we already handle—less important. We do at least provide inter-experiment resource isolation since physical nodes are dedicated to experiments, hosting virtual nodes only for that experiment. Finally, we hoped to achieve at least a ten fold multiplexing factor on low-end PCs (850 MHz, 512 MB memory), necessitating a lightweight virtualization mechanism. Considering these requirements, a process-level virtualization seemed the best match. Given our BSD heritage and expertise, we opted to design and implement our virtual nodes by extending FreeBSD jails. In the following discussion, we refer to an instance of our virtual node implementation as a *vnode*.

Jails. Jails provide filesystem and network namespace isolation and some degree of superuser privilege restriction. A jailed process and all its descendents are restricted to a unique slice of the filesystem namespace using *chroot*. This not only gives each jail a custom, virtual root filesystem but also insulates them from the filesystem activities of others. Jails also provide the mechanism for virtualizing and restricting access to the network. When a jail is created, it is given a virtual host-name and a set of IP addresses that it can bind to (the base jail implementation allowed a single IP address with a jail, we added the ability to specify multiple IP addresses). These IP addresses are associated with network interfaces outside of the jail context and cannot be changed from within the jail. Hence, jails are implicitly limited to a set of interfaces they may use. We further extended jails to correctly limit the binding of the IN-ADDR_ANY wildcard address to only those interfaces visible to the jail and added restricted support for raw sockets. Finally, jails allow processes within them to run with diminished root privilege. With root inside a jail, applications can add, modify and remove whatever files they want (except for device special files), bind to privileged ports, and kill any other processes in the same jail. However, jail root cannot perform operations that affect the global state of the host machine (e.g., reboot).

Virtual disks. Our design of virtual disks made it easy not only to be efficient in disk use, but also to support inter-vnode disk space separation. Jails provide little help: even though each jail has its own subset of the filesystem name space, that space is likely to be part of a larger filesystem. Jails themselves do nothing to limit how much disk space can be used within the hosting filesystem.

Our design uses the BSD *vd* device to create a regular file with a fixed size and expose it via a disk interface. Filesystem space is only required for blocks that are allocated in a virtual disk, thus this method is space-efficient for the typical case where the virtual disk remains mostly empty. These fixed-size virtual disks contain a root filesystem for each jail, mounted at the root of each jail's name space. Since the virtual disks are contained in regular files, they are easy and efficient to move or clone.

Control of vnodes. While enhancing the Emulab system with node types other than physical cluster nodes, we worked to preserve uniformity and transparency between the different node types wherever possible. The result is that the system is almost always able to treat a node the same, regardless of its type, except at the layers that come in direct contact with unavoidable differences between node types, or when we aggregate expensive actions by operating through the parent physical node.

An example of the transparency is the state machines

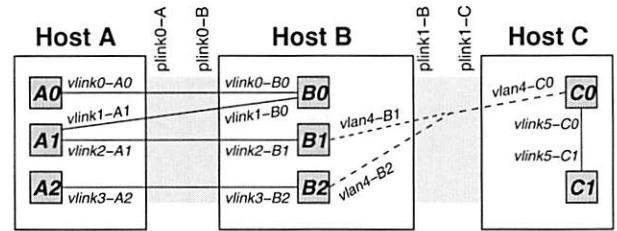


Figure 1: A network topology illustrating routing issues due to the multiplexing of virtual nodes and links. Large boxes represent physical nodes and links, while small boxes and lines (with *italic labels*) represent virtual nodes and links. Virtual network interfaces (*vlinks*), virtual LANs (*vlan*s), and physical links (*plinks*) have names as shown.

used to monitor and control nodes of all types. While non-physical nodes have significant differences from physical nodes, the state machines used to manage them are almost identical. In addition, the same machine is used for Emulab vnodes as well as PlanetLab virtual servers. Reusing—indeed, sharing—such complex and crucial code contributes to the overall system's reliability.

3.2 Virtual Networks

Virtualizing a network includes not only multiplexing many logical links onto a smaller number of physical links, but also dealing with network namespace isolation issues and the subtleties of interconnecting virtual nodes within, and between, physical nodes. As with virtualizing nodes, there is a range of techniques available for virtualizing networks. When choosing the technology for Emulab, the criteria we evaluated against were:

Level of virtualization. A virtual network implementation can present either a virtual layer 2 (e.g., Ethernet) or a virtual layer 3 (e.g., IP).

Use of encapsulation. Virtual network links may (e.g., 802.1Q VLANs) or may not (e.g., “fake” MAC addresses) encapsulate their packets when traversing physical links.

Sharing of interfaces. The end point of a virtual network link as seen by a virtual node may be either a shared physical interface device or a private virtual one. This may affect whether interface-centric applications like *tcpdump* can be used in a virtual node.

Ability to co-locate virtual nodes. Can there be more than one virtual node from a given network topology on the same physical host? If so, there are additional requirements for correct virtualization.

3.2.1 Emulab Virtual Networks

The Emulab virtual node implementation uses unshared, virtual Ethernet devices in order to maintain transparency with the “bare machine” model which presents dedicated physical Ethernet devices to applications. By default, these virtual devices are configured to use a custom 16-byte encapsulation format, allowing use of the virtual devices with any switching infrastructure. There is also an option to allow rewriting the source MAC address in outgoing packets with the virtual MAC address. Since any physical host is dedicated to a single experiment, whether using virtual nodes or not, it is necessarily the case that virtual nodes for a topology will be co-located. This raises two issues related to forwarding packets that are addressed in the next sections.

Virtual network interfaces. While the FreeBSD jail mechanism does provide some degree of network virtualization by limiting network access to specific IP addresses, it falls short of what we need. In particular, though jails have their own distinct IP addresses, those IP addresses are associated directly with shared physical interfaces, and thus have problems with interface-oriented applications such as `tcpdump`. Moreover, because we allow co-location, it is possible that two virtual nodes in the same LAN could wind up on the same physical host, for example B1 and B2 in Figure 1. FreeBSD does not allow two addresses in the same subnet to be assigned to one interface.

To solve these problems, we developed a virtual Ethernet interface device (“veth”). The veth driver is an unusual hybrid of a virtual interface device, an encapsulating device and a bridging device. It allows us to create unbounded numbers of Ethernet interfaces (virtualization), multiplex them on physical interfaces or tie them together in a loopback fashion (bridging) and have them communicate transparently through our switch fabric (encapsulation). These devices also provide the handle to which we attach the IPFW/Dummynet rules necessary for doing traffic shaping. Veth devices can be bridged with each other and with physical interfaces to create intra- and inter-node topologies and ensure the correct routing of packets at the link level. For example, this bridging prevents “short-circuit” delivery of traffic between co-located nodes A0 and A2 in Figure 1, which might otherwise occur when FreeBSD recognized that both interfaces are local. Since multiple veth devices may be bridged to the same physical device, incoming packets on that device are demultiplexed based on the virtual device’s MAC address, which is either contained in the packet (encapsulation) or is exposed directly as the packet’s MAC address (no encapsulation).

Virtual routing tables. While virtual Ethernet devices are sufficient to enable construction of virtual Eth-

ernet topologies, they are not sufficient to support arbitrary IP topologies. This is due to FreeBSD jails sharing the host’s IP infrastructure, in particular, the routing table. In the routing table, it is only possible to have one entry per destination. But with a physical node hosting multiple jails representing different virtual nodes at different points in the topology, we need to be able to support multiple routes to (next hops for) a single destination. This is known as the “revisitation” problem [24]. For example, in Figure 1, packets sent from A0 to C0 will pass through host B twice. B0’s next hop for C needs to be A (for A1) while B1’s needs to be C (for C0). Thus there need to be separate routing tables for B0 and B1. Further, even with separate routing tables, incoming packets to B need context to determine which routing table to use.

For Emulab, we have adopted and extended the work of Scandariato and Risso [22] which implements multiple IP routing tables to support multiple VPN end points on a physical node. Routing tables are identified by a small integer routing table ID. An ID is the glue that binds together a jail, its virtual interfaces, and a routing table. Incoming packets for different jails on a physical node can thus be differentiated by the ID of the receiving interface and can be routed differently based on the content of the associated routing table.

3.2.2 IP Address Assignment

A subtle aspect of implementing virtual networks is assigning addresses, in particular IPv4 addresses, to the potentially thousands of links which make up a topology. The topologies submitted to Emulab typically do not come annotated with IP addresses; most topology generators do not provide them, and it is cumbersome and error-prone for experimenters to assign them manually. We thus require an automated method for producing “good” IP address assignments, and it must scale to the large networks enabled by virtualization. A desirable address assignment is one that is realistic—that is similar to how addresses would be allocated in a real network. In the real world, the primary (though not only) factor that influences address assignment is the underlying hierarchy of the network. Hierarchical address assignment also leads to smaller routing tables and thus better scaling. Since real topologies are not strictly hierarchical, the challenge becomes identifying a suitable hierarchical embedding of the topology. Our work on IP address assignment centers on inferring hierarchy in this practical setting.

We developed and evaluated three different classes of algorithms: bottom-up, top-down, and spectral methods, described in detail elsewhere [8]. The approach we ultimately deployed in production, called recursive parti-

tioning, creates the IP address tree in a top-down manner. At the top level, the root of the tree contains every node in the graph. Then we partition it into two pieces using a graph-partitioner [17], assigning each half of the graph to a child of the root. By applying this strategy recursively, we create a tree suitable for IP address assignment. The result is a fast algorithm that produces small routing tables: for example, it can assign addresses to networks of 5000 routers—comparable to today’s largest single-owner networks—in less than 3 seconds.

4 Automated Resource Assignment

When an experimenter submits an experiment, Emulab automatically chooses a set of physical nodes on which to instantiate that experiment. This process of mapping the virtual topology to a physical topology is called the network testbed mapping problem [21], and virtual nodes add new challenges to an already NP-hard problem. The needs of this mapping are fundamentally similar to other virtualized networking environments, such as the planned GENI facility [10] (where such mapping will be done by a Slice Embedding Service), and Model-Net [26].

For an experiment with virtual nodes, a good mapping is one that “packs” virtual hosts, routers, and links on to a minimum number of physical nodes without overloading the physical nodes. This means, for example, placing, when possible, nodes that are adjacent in the virtual topology on the same physical node, so that the links between them need not use physical interfaces or switch capacity. This is particularly difficult because the virtual nodes may not have uniform resource needs, and physical nodes may not have identical capacities. Since Emulab is a space-shared testbed, it is also important that bottleneck resources, such as trunk links between switches, are conserved, since they may be needed by other concurrent experiments. Finally, experimenters may request nodes with special hardware or software, and the mapper must satisfy these requests.

Emulab finds an approximate solution to the network testbed mapping problem by taking a combinatorial optimization approach. It uses a complex solver called *assign* [21]. *assign* is built around a simulated annealing core: it uses a randomized heuristic to explore the solution space, scoring potential mappings based on how well they match the experimenter’s request, avoid overloading nodes and links, and conserve bottleneck resources. We found, however, that Emulab’s existing *assign* was not sufficient for mapping virtual node experiments, and enhanced it accordingly.

First, we needed new flexibility in specifying how virtual nodes are to be multiplexed (“packed”) onto physical nodes. To get efficient use of resources, we found it nec-

essary to add fine-grained resource descriptions, and to relax *assign*’s conservative resource allocation policies.

Second, because virtualization allows for topologies that are an order of magnitude larger than one-to-one emulation, we ran into scaling problems with *assign*. Since it must be run every time an experiment is swapped in or re-mapped as part of auto-adaptation, runtimes in the tens of minutes were interfering with the usability of the system and making auto-adaptation too cumbersome. To combat this, we made enhancements to *assign* that exploit the natural structure of the virtual topologies it is given to map.

4.1 Flexible Resource Specification

assign must use some criteria to determine how densely it can pack virtual nodes onto physical nodes. *assign* already had the ability to use a coarse-grained packing, in which each physical node has a specified number of “slots,” and each virtual node is assumed to occupy a single slot. Thus, it can be specified that *assign* may pack up to, for example, 20 virtual nodes on each physical node. It became clear that this would not be sufficiently fine-grained for many applications, including our auto-adaptation scheme, because different virtual nodes will have different roles in the experiment, and thus consume different amounts of resources.

To address this, we added more packing schemes to *assign*. In the first, virtual nodes can fill more than one slot; experimenters can use this when they have an intuitive knowledge, for example, that servers in their topology will require more resources than clients by an integer ratio: 2:1, 10:1, etc.

The second packing scheme models multiple independent resources such as CPU cycles and memory, and can be used when the experimenter has estimated or measured values for the resource needs of the virtual nodes. Each virtual node is tagged with the amount of each resource that it is estimated to consume, and *assign* ensures that the sum of resource needs for all virtual nodes assigned to a particular physical node does not exceed the capacity of the physical node. This scheme builds on *assign*’s system of “features and desires”: virtual nodes can be identified as having “desires” which must be matched by “features” on the physical nodes they are mapped to. Features and desires are simply opaque strings, making this system flexible and extensible. We have enhanced *assign* to allow features and desires to also express capacities, which are then enforced as described above. While we use this scheme for relatively low-level resources (CPU and memory), it could also be used for higher-level metrics such as sustainable event rate for discrete event simulators such as *ns*.

The resource-modeling scheme is particularly useful

for feedback-based auto-adaptation. The values used for CPU and memory consumption of a virtual node can simply be obtained by taking measurements of an earlier run of the application. The maximum or steady-state usage can then be used as input to the mapping process. The coarse-grained and resource-based packing criteria can be used in any combination.

In addition to packing nodes, virtual links must be packed onto physical links. Though the two types of packing are conceptually similar, a different set of issues applies to link packing. Some of these issues exist for one-to-one emulation, but there are also some new challenges that come with virtual emulation.

Link mapping issues that one-to-one and virtual emulation have in common. First, physical nodes in a Emulab-based testbed have multiple interfaces onto which the virtual links must be packed. Second, the topology of the experimental network is typically large enough that it is comprised of multiple switches. These switches are connected with links that become a bottleneck, so the mapping must be careful to avoid over-using them.

Link mapping challenges that arise with virtual emulation. When mapping virtual-node experiments, links between two virtual nodes that are mapped to the same physical node become “intra-node” links that are carried over the node’s “loopback” interface. It is advantageous to use intra-node links, as they do not consume the limited physical interfaces of the physical node. Although the bandwidth on a loopback interface is high, there are practical limits on it, and for some experiments that use little CPU time but large amounts of bandwidth, loopback bandwidth can become the limiting factor for packing virtual nodes onto physical ones. We have extended `assign` to take this finite resource into account.

One of the guiding principles of `assign` has historically been conservative resource allocation; when assigning links, it ensures that the full bandwidth specified for the link will always be available. While this makes sense for artifact-free emulation, is at odds with our goal of using virtualization to provide best-effort, large-scale emulation. For example, an experimenter may have a topology containing a cluster of nodes connected in a LAN. Though the native speed of this LAN is 100 Mbps, the nodes in this LAN may never transmit data at the full line rate. Thus, if `assign` were to allocate the full 100 Mbps for the LAN, much of that bandwidth would be wasted. To make more efficient resource utilization possible, we have added a mechanism so that estimated or measured bandwidths can be passed to `assign`. As with node resources, this bandwidth can be measured as part of auto-adaptation.

4.2 Improving `assign`’s Scaling

`assign` has been designed and tuned to run well on Emulab’s typical one-to-one workload, consisting of topologies with at most a few hundred nodes. In order to make `assign` scale to topologies of the scale enabled by virtual nodes, we developed several new techniques.

4.2.1 Searching the Solution Space

Our first techniques for tackling scaling issues are aimed at improving the way in which `assign` searches through the solution space of possible mappings. `assign` finds sets of homogeneous physical nodes and combines them into equivalence classes; this allows it to avoid large portions of the solution space which are equivalent, and thus do not need to be searched. However, this strategy breaks down with the high degree of multiplexing that comes with virtual-node experiments, because a physical node that has been partially filled is no longer equivalent to an empty node. We have addressed this problem by making these equivalence classes adapt dynamically at run time, with physical nodes entering and leaving classes as virtual nodes are assigned or unassigned to them.

Another improvement to the search strategy came from the observation that, in a good solution, nodes that are adjacent in the virtual topology will tend to be placed on the same physical node. So, we made an enhancement to the way `assign` selects new virtual-to-physical mappings to try, as it moves through the search space. To conduct this search, `assign` takes a potential solution, selects a virtual node, selects a new physical node to map it to, and determines whether or not the resulting mapping is better than the original. This process is repeated, typically hundreds of thousands or millions of times, until a no better solutions are found. In our modified version, rather than selecting a random physical node, with some probability, `assign` selects a physical node that one of the virtual node’s neighbors has already been mapped to. This improvement made a dramatic difference in solution quality, leading to much tighter packing and exhibiting much better behavior in clustering connected nodes together.

4.2.2 Coarsening the Virtual Graph

Though these changes to the search strategy improved `assign`’s runtime and solution quality, running `assign` on very large topologies could still take more than an hour, much too long for our purposes. To make the problem more tractable, we exploit topological features of the virtual topology.

We expect that most large virtual topologies will be based on the structure of the Internet; these may come

from actual Internet “maps” from tools like Rocket-fuel [23] or from topology generators designed to create Internet-like networks, such as GT-ITM [33], inet [29], and Orbis [15]. The key realization is that such networks tend to have subgraphs of well-connected nodes, such as ISPs, ASes, and enterprises. In addition, we expect that many topologies will have edge-LANs that represent clusters, groups of workstations, etc.

We exploit the structure of the input topology by applying a heuristic coarsening pre-pass to the virtual graph before running *assign*. By giving *assign* a smaller virtual topology, we reduce the solution space that it must search, in turn reducing the time required to find a good solution. The goal of this pre-pass is to find sets of virtual nodes that, in a good mapping, will likely be placed on a single physical node. A new virtual graph is then generated, with each of these sets combined into a single node. These “conglomerates” retain all properties of their constituent nodes; for example, the CPU needs of each constituent are summed together to produce the CPU required for the conglomerate.

We have implemented two coarsening algorithms. The first stems from the realization that many topologies contain LANs representing groups of clients or farms of servers. An optimal mapping will almost always place as many members of these LANs onto a single physical node as possible. So, we find leaf nodes in LANs (that is, nodes whose only network interface is in that LAN), and combine all leaf nodes from the same LAN into a conglomerate.

The second algorithm uses a graph partitioner, METIS [17], to partition the nodes in the virtual graph. We choose a number of partitions such that the average partition will fit on the “smallest” available physical node. We then combine the virtual nodes in each partition into a single conglomerate node. The quality of the partitions returned by the partitioner is dependent on the extent to which separable clusters of nodes are present in the graph. Since we are focusing on Internet-like topologies with some inherent hierarchy, we expect good results from this method.

The coarsening algorithms (particularly METIS) do not know the intricacies of the network testbed mapping problem, such as constraints on node types, resource usage, and link bandwidths; this is one reason they are able to run much faster than *assign* itself. This leaves us with the problem that they may return sets of nodes to cluster that cannot be mapped onto any physical resources; for example, they may require too much CPU power or have more bandwidth than a single node can handle. Once the coarsening algorithm has returned sets of nodes, we use a multidimensional bin-packing approximation algorithm to pack these into the minimum number of mappable conglomerates.

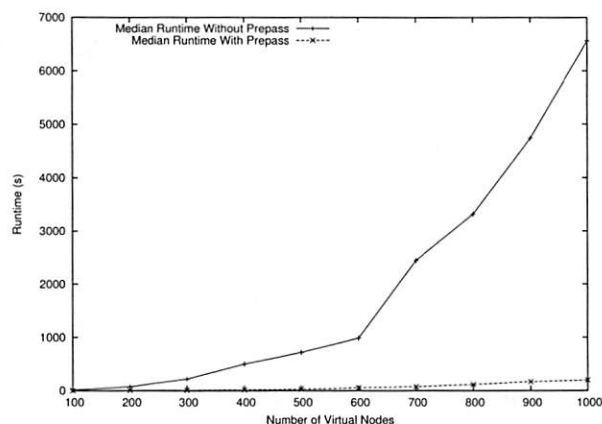


Figure 2: Median runtime of *assign* with and without a coarsening pre-pass.

Both coarsening algorithms help *assign* to run faster by making heuristic decisions that limit *assign*’s search space, but could, in turn, make clustering decisions that result in sub-optimal mapping. However, in our domain obtaining a solution in reasonable time is more important than obtaining a near-optimal solution. The mappings obtained by *assign* will always be valid, but it is possible that some topologies are coarsened in such a way the mapping does not make the most efficient use of resources. The biggest potential problem is fragmentation, in which the coarsening pass makes conglomerates whose sizes do not pack well into the physical nodes. We take measures to try to avoid this circumstance, by carefully choosing our target conglomerate size. In practice, the worst fragmentation we have seen caused only a 13% increase in physical resources used.

To evaluate our new resource mapper as well as to understand the effects of the coarsening pre-pass, we compared runs of *assign* with and without the pre-pass. These runs mapped transit-stub topologies generated by GT-ITM [33] onto Emulab’s physical topology. Each test was run ten times. In all cases, the runtime of the pre-pass itself was negligible compared to the runtime of *assign*.

Figure 2 presents the median runtimes for these tests on a 1.5 GHz Pentium IV, showing the greatly significant time savings from the pre-pass. As we scale up the number of virtual nodes the improvement goes from a factor of 15 at 100 nodes (12.0 to 0.78 seconds), to a factor of 32 at 1000 nodes (6560 to 200 seconds). The absolute result is also good: it takes just 200 seconds to map 1000 nodes.

This speedup, of course, does not come without a cost. Figure 3 shows the decrease in solution quality, in terms of the quality of link mappings. Intra-node links connect two virtual nodes mapped to the same physical node;

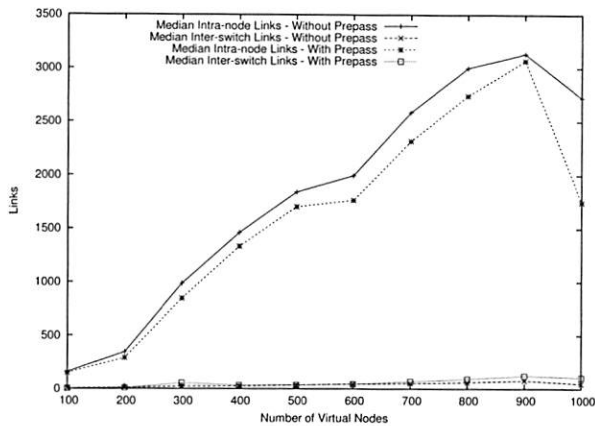


Figure 3: Number of intra-node and inter-switch links found by assign. Larger numbers of intra-node links are better, and smaller numbers of inter-switch links are better.

they do not use up shared switch resources, so having a large number of them is an indicator of a good mapping. Inter-switch links, on the other hand, are an indicator of a poor mapping, because they consume the shared resource of inter-switch links. Though the pre-pass does cause assign to find somewhat worse mappings, the differences are tolerable, and the speedup is a clear win. In over 70% of the test cases, the number of intra-node links found when using the pre-pass was within 10% of the number found by assign by itself. The worst run was within 16%.

5 Exploiting Physical Hierarchy

In addition to the previously described IP assignment and mapping problems, a number of more general but severe “system” scaling issues arose, which prevented us from reaching large size until we addressed them. Some are system-wide issues that are the byproducts of the order of magnitude increase in the potential size of an experiment. Others are per-node issues that are the result of increasing the resource consumption on a node. In both cases, we devise solutions that exploit the physical structure and realities of the physical testbed infrastructure.

Most system-wide problems have to do with accessing centralized services and the use of unreliable protocols, primarily during initial experiment setup. The system-wide scaling problems encountered here are essentially the same issues faced when increasing the number of physical machines in the testbed. For example, sharing a single NFS filesystem does not scale well. We are constantly addressing these types of issues as we expand into larger virtual node experiments. Ultimately, virtual node growth will continue to outpace physical resource growth by 1–2 orders of magnitude. However, by leveraging the close relationship between virtual nodes and their host

we significantly reduce the burden on the central infrastructure as highlighted by the following examples.

There are a number of situations in which we use the physical host as a caching proxy for its hosted virtual nodes. Nodes in Emulab “self configure” when they boot, after obtaining the necessary configuration information from a central server. Since the physical host necessarily boots before its virtual nodes, it downloads configuration information for all virtual nodes in a single operation, pre-loading a cache for each, and in some cases, performing configuration operations itself in a more efficient manner. Similarly, the physical host acts as an Emulab event system proxy, using a single connection to the master event server to collect and distribute control events for all its virtual nodes.

One of the most compute-intensive parts of instantiating an experiment is calculating routing tables for all of the nodes. Though Emulab supports dynamic routing through the use of a routing daemon such as gated or zebra, most experimenters prefer the consistency and stability offered by computing routing tables off-line before the experiment begins. Typical algorithms for doing this, however, have runtimes ranging from $O(V^2 \cdot \lg(V) + V \cdot E)$ (Dijkstra’s algorithm with a Fibonacci heap) to $O(V^3)$ (Dijkstra’s algorithm with a linear-array priority queue), with respect to the number of vertices (nodes) and edges (links) in the topology graph. To solve this problem, we parallelize route computation across all of the physical nodes in the experiment, with each physical node responsible for the routing tables of the virtual nodes it hosts. We distribute one copy of the topology to each physical host, and run Dijkstra’s algorithm sourced from each virtual node hosted on that physical node. Thus the route calculation time becomes $O(V^2 \cdot n)$, where n is the number of virtual nodes hosted on each physical node. In practice, with the size of virtual topologies that are feasible to run on Emulab and the level of virtual-to-physical multiplexing possible, this time never exceeds a few seconds.

The original Emulab system could not reliably instantiate an experiment larger than about 100 nodes. Our improvements in Emulab allow experiments of up to at least two thousand nodes to be reliably instantiated. A fundamental limitation on speed of instantiation is that vnode construction is not parallelizable within a uniprocessor host. However, virtual nodes on distinct physical hosts can be setup in parallel. To demonstrate the degree to which this parallelism can be successfully exploited, we performed a simple test in which an experiment consisting of a single LAN was repeatedly instantiated, each time adding to the LAN one physical node hosting 10 virtual nodes. In the base case of one physical node with 10 virtual nodes in the LAN, setup, including topology mapping, node configuration and startup, required 194

seconds. At 80 virtual nodes on 8 physical nodes, it took 290 seconds, a 50% increase in time for an 800% increase in size.

6 Feedback-Directed Resource Allocation

Maximum scalability is achieved when Emulab's physical nodes and networks can be divided as finely as possible, each physical resource providing support to as many emulated and/or simulated entities as possible. However, for these emulated and simulated environments to be worthwhile to most Emulab users, they must be accurate recreations of devices in the real world. Meeting our scalability goal and our realism constraint at the same time means making virtual nodes that are "just real enough" from the point of view of the software systems under test.

Finding the proper balance between scalability and fidelity is not easy: the ideal tradeoff that is "just real enough" is inherently specific to the software being tested. Therefore, to find the appropriate resource mappings for a user's experiment, our technique is to automatically search for a mapping that minimizes physical resource use while preserving fidelity according to application-independent (provided by the system) and/or application-dependent (provided by the user) feedback.

Testbed users have two options for adapting their experiments: executing a single-stage "training" run that requires little effort, or running a multi-stage *automatic experiment adapter* that requires additional effort.

The first option does not require the experiment to be fully automated, so is suitable for an interactive style of experimentation. In this model, the user creates an experiment and swaps it in on virtual nodes mapped one-to-one on physical nodes to ensure adequate resources. Users can then login to the nodes, run their programs, and, when they have determined that the experiment is in a representative state, click a button to record a profile. This profile is then used in subsequent runs to drive the resource mapping. Of course, because of the one-to-one initial mapping, this simplistic manual approach will not work for large topologies. For those topologies we will necessarily need to start out with some virtual nodes multiplexed many to one on physical nodes and thus we cannot gather an accurate resource requirements profile with a single run. For these situations we offer the second option.

In the multi-stage approach, an experiment is automatically run multiple times, each time adjusting the mapping to account for any resource overloads noted in the previous run. To do this, the user must automate the execution of the experiment. Each run of the experiment starts up a representative workload, monitors resource usage once that workload reaches a steady state, invokes

a script to gather the monitor output and create a profile, and then remaps and reinstantiates the experiment based on that profile. This process continues until there is a run in which no resource overload is detected.

Our feedback-driven adaptation technique automatically finds virtual-to-physical mappings that provide the user's required level of emulation fidelity while allowing Emulab to make maximally efficient use of its resources. There is a risk, however, that the mappings set up by the adapter will fail to provide sufficient fidelity to the user's software during "production" testbed runs, e.g., because the user modifies the software or is driving it in a different way. Emulab relies on run-time feedback to detect such cases and signal the user about possible problems with his or her experiment.

6.1 Implementation

Ensuring application fidelity when multiplexing virtual nodes can be achieved quickly and accurately through monitoring the application's steady state resource usage and feeding this data back into *assign*. Utilizing application-independent metrics, like CPU and memory usage, we can automatically adapt the packing of virtual resources on to physical hosts. This is done in a way that minimizes physical resource use while leaving sufficient headroom for the vnode's steady state resource consumption. Any available application-specific metrics can then be used to refine the mapping to account for lack of precision in the low level data.

On each physical node Emulab gathers a number of application-independent resource usage statistics to feed back to the adaptation mechanism. These include CPU use, interrupt load, disk activity, network traffic rates, and memory consumption. CPU and memory information are also gathered at vnode granularity, which is how we determine the resource demand of individual vnodes. The other global statistics allow us to ensure that the physical node as a whole is not overloaded.

The multi-stage adaptation technique requires that an experiment be automated, running a particular sequence of actions. This is easily done using Emulab's event system, which allows for executing operations in parallel, in sequence, or at specific times relative to the start of the experiment. The user only needs to create an event sequence to perform the steps described earlier. There are built-in events for two feedback specific activities. One allows for running the application-independent resource monitor on all nodes for a fixed length of time. The other performs the remap itself, a process which includes gathering the log files from all nodes, analyzing the data to detect overloads and produce new per-node resource usage estimates, and finally invoking *assign* and reconfiguring the experiment to reflect the new virtual node

layout.

The mechanism for supporting application-dependent metrics and providing user-directed feedback based on those metrics is a prototype and likely to change. In the current implementation, the user must provide three components. First, he or she provides one or more resource monitors that gather and log appropriate resource usage data. These could be separate programs, or they could just be the applications themselves, logging relevant information. Second, the user provides a “baseline summary” file describing the expected behavior of the system when not constrained by node resources. This summary file can be in any format, as it is interpreted by a user-supplied script. That script is the third component. The script is automatically invoked at the end of each run of the experiment, with the log files from the monitors and the baseline summary as inputs. Its job is to aggregate the log file information into a new summary and compare that summary with the baseline to determine if there is a resource overload. If the answer is “yes,” then the experiment will be remapped.

6.2 Usage Scenarios

We see three common ways in which the auto-adaptation mechanism can be used. In the first, the user starts with a one-to-one mapping of virtual nodes to physical nodes and “packs” the experiment into fewer physical nodes. Starting with the one-to-one mapping we gather bootstrap resource data in the first pass. The system then runs successive passes, increasing or decreasing the packing until it arrives at a maximally dense packing factor with virtual node resource use that is consistent with the one-to-one mapping. At this point, the user will probably want to increase the size of her or her topology. The simplest approach is to use the bootstrap data for nodes that will remain in the experiment and perform a bootstrap on the newly added nodes. Alternatively, the user can divide nodes into resource classes (e.g., client/server), which are initialized using data derived from previous runs.

A second style of adaptation, using the same mechanism, is to start with a dense mapping of a topology and then expand it. A dense mapping is achieved by providing no initial feedback data, allowing *assign* to map strictly on the basis of available physical node and link characteristics. In this configuration, there can be no training run to gather clean resource usage data. Instead, feedback data are provided by the application-independent metrics (pushing the experiment away from obvious overload conditions) or with interactive guidance from the user. This form of adaptation is used with large topologies where there are not enough physical resources to map it one-to-one.

Finally, in the third scenario, an Emulab experiment can incorporate purely simulated nodes and networks, using a modified version of *nse* [9]. As described in detail elsewhere [12, 13], these simulated entities can be transparently spread across physical nodes, just as *vnodes* are dispersed. Since these simulated nodes interact with real traffic, the simulator must keep up with real time. Detecting when virtual time has significantly fallen behind real time gives us a way to detect overload that is more straightforward than with *vnodes*, although there are subtleties with synchronizing the two that must be taken into consideration, as described in the above references. Our infrastructure can adaptively remap simulated networks similarly to the way it handles virtualized nodes and links.

7 Results

In the following sections we present a preliminary evaluation of the Emulab virtual node implementation in three areas: application fidelity, application transparency, and performance and fidelity of the adaptation mechanism. All results were gathered on Emulab’s low-end “pc850” machines, 850 MHz PCs with 512 MB of RAM and four 100 Mb Ethernet interfaces.

7.1 Application Fidelity

Microbenchmarks. To get a lower-level view of fidelity with increasing co-location, we performed an experiment in which we ran the *pathrate* [7] bandwidth-measurement tool between pairs of nodes co-located on the same physical host. Each pair of nodes was connected with a T1-speed (1.5 Mbps) link. We measured the bandwidth found by *pathrate* as we increased the number of node pairs from one to ten. Across all runs, *pathrate* measured the correct bandwidth to within 1 Kbps, with a standard deviation across runs of *pathrate* of 0.004.

Applications. We ran a synthetic peer-to-peer file sharing application called *Kindex*, which is modeled after a peer music file sharing network such as KaZaa. *Kindex* maintains a distributed peer-to-peer index of file contents among a collection of peer servers. It also keeps track of replicas of a file among peers and their proximity, to expedite subsequent downloads of the same file. In our simplified experiment, we start a series of 60 clients sequentially. Each of 60 clients uploads a single file’s information to the global index, and starts randomly searching for other files, fetching those not previously fetched into its local disk. Each client generates between 20 to 40 requests per minute for files, whose popularity follows a Zipf distribution. Each client has sufficient space to hold all 60 files. Hence after the experiment has run for

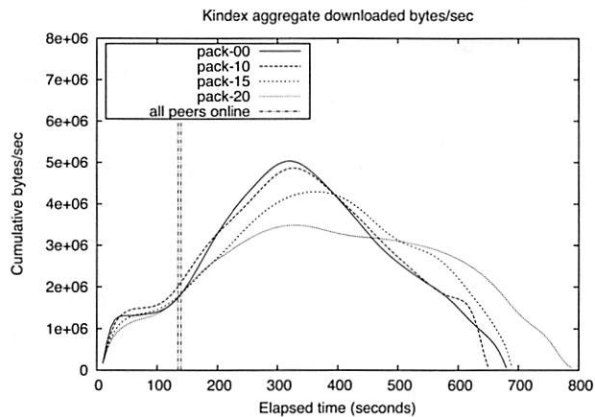


Figure 4: Cumulative system bandwidth for co-location factors of 0, 10, 15 and 20. “All peers online” is the point in time where all 60 peers are running and downloading files.

a while, all clients end up caching all files, at which time we stop the experiment.

The network topology consists of six 10 Mbps campus LANs connected to a core 40 Mbps LAN of routers with 100 ms roundtrip between themselves. Each campus LAN is connected to a router via a 3 Mbps, 20 ms RTT link.

We plotted the aggregate bandwidth delivered by the system to all its users as a time line. For this, we measured the total size of files downloaded by all users in every 10-second interval. We expect that initially downloads are slow, but as popular files are cached widely, subsequent downloads are more likely to be satisfied from a peer within the same campus, driving up the aggregate bandwidth due to the higher speed links. However, due to the fetch-once behavior of clients, as more files are downloaded by all users, downloads become less frequent, driving down the aggregate bandwidth.

We ran the experiment in four configurations. First, we emulated the topology on just physical nodes to establish a base line. We then repeated the experiment using virtual nodes with co-location factors of 10, 15 and 20 virtual nodes per physical node. Figure 4 shows the results. The base line (pack-00) shows the expected behavior, aggregate bandwidth increasing to a peak and then tapering off. At a co-location factor of 10, one campus LAN mapped per physical node, the behavior is indistinguishable for the base line. However, as we increase the co-location to 15 and 20, since peers have to supply files over the faster LAN links, the load on the local disk rises. This is the reason for the reduced peak bandwidth and its shift to the right, causing the curve to be flattened.

While this example shows that we can achieve an order of magnitude scaling improvement with an IO intensive

application on low-end PCs, it also illustrates the utility of feedback data for driving virtual node multiplexing. In this example, some node disks hit 100% busy in both the pack-15 and pack-20 cases, an event easily detected by application-independent metrics. However, the user might also decide that the results from pack-15 were acceptable, but those in pack-20 were not. In this case they might construct a custom metric saying that remapping is only necessary if the disk were saturated for three consecutive measurements.

7.2 Application Transparency

Correctly achieved transparency is difficult to rigorously demonstrate; only failures of transparency are obvious. Our most compelling evidence is that experimenters have run thousands of diverse virtual node experiments, yet generated only a handful of requests for “missing features” such as support for multicast routing and IPFW firewall rules. We did perform one empirical stress test, running a routing daemon in a complex virtual network topology. By causing a series of link failures within the topology, we verified that the routing daemons were functioning as expected. In that test, we ran unmodified gated routing daemons on all nodes in a 416 vnode hierarchical topology on 22 PCs and automatically generated OSPF configuration scripts. Once we verified the connectivity between some leaf nodes across the diameter of the topology, we caused a link failure in the interior to see how OSPF would route around the failure. Before the failure, a route between two leaf nodes was symmetric with 11 hops. We found a 5 second downtime in one direction and 9 seconds in the reverse direction, after which alternate 12 hop paths were established. The forward and reverse paths were different in one hop. When we removed the link failure, it took 22 and 28 seconds respectively for the route paths to be restored. Finally, we rebooted two interior nodes in the topology. gated restored all the routes in a little over a minute.

7.3 Adaptation Results

We evaluated our feedback system in three scenarios: a Java-based web server and clients, the BitTorrent peer-to-peer file distribution system, and the Darwin Streaming Server [2].

We first ran a Java-based web server on one host with 69 clients continually downloading a 64 KB file. The clients were separated into three different types based on their link characteristics. Nine clients were evenly spread across three links on a single router using 2 Mb LANs to emulate cable modem clients. Forty clients were directly connected to a single router using 2 Mb multiplexed links to emulate DSL modems. Finally, 20 clients

Metric	2 Mb LAN	2 Mb Link	56 Kb Link
74 vnodes on 74 physical nodes			
Avg. Transaction Rate	1.19	2.29	0.09
Avg. Response Time (s)	0.84	0.43	10.67
Packed onto 7 phys. nodes after first iteration			
Avg. Transaction Rate	1.10	1.85	0.09
Avg. Response Time (s)	0.91	0.53	10.77
Packed onto 7 phys. nodes after second iterations			
Avg. Transaction Rate	1.19	2.29	0.09
Avg. Response Time (s)	0.84	0.43	10.70

Table 1: Performance of clients continually downloading a 64 KB file in different vnode mappings.

were directly connected to a single router using 56 Kb multiplexed links, to emulate phone modem clients. The feedback loop required three iterations to reach acceptable application fidelity; the results are shown in Table 1. The first iteration is a one-to-one mapping that allows the system to get a clean set of feedback data. The second iteration packed the 74 vnodes onto 7 physical nodes and resulted in a drop in performance because the CPU intensive server node was co-located with several client nodes. The final iteration amplifies the feedback data (i.e., increases the CPU and memory requirements) by 20%, which is enough to isolate the server and return the application metrics to their original one-to-one values, without allocating any more physical nodes. It should be noted that the bad mapping found in the second iteration could have been avoided with higher precision monitoring. However, in our context a bad initial remapping is a benefit because it denotes the lower bound on the number of required nodes and we always wish to minimize the number of physical nodes required for a topology.

To demonstrate scaling a real application to large topologies that cannot fit in a one-to-one mapping, we ran the BitTorrent p2p file distribution program on a 310-node network packed onto 74 physical nodes. The topology consisted of 300 clients communicating over 2 Mb LANs or links, a single “seed” node with a 100 Mb link, and nine routers that formed the core. To bootstrap the mapping we used feedback data from a smaller topology for the clients, since their resource usage was dependent on the link constraints and not the number of clients in the system. However, the resource use of the seed node and routers is tied to the size of the network, so they were left one-to-one. In total, it took 19 minutes to instantiate the topology: seven minutes for assign to map the virtual topology onto the physical topology and twelve minutes to load disks onto the machines, reboot, and setup the individual virtual nodes. This should be comparable to the length of time it would take to setup the same

Mapping	Video gap (ms)		Audio gap (ms)	
	Min	Max	Min	Max
One to one	0.93	90.99	48.23	210.96
Phys. Link Shared	0.04	470.3	0.07	531.27
Phys. Link Unshared	0.54	91.99	30.88	232.10

Table 2: Interpacket gap of clients receiving a 100 Kbps video and audio stream in different configurations where the physical link is shared and not shared. The values are the median of five runs.

topology on physical machines (if Emulab had sufficient nodes). On physical nodes, the assign time would be less, but the time to setup switch VLANs would exceed the time required to setup virtual links.

The adaptation mechanism can also accommodate applications that have throughput constraints as well as timing sensitivity. We tested the Darwin Streaming Server sending a 100 Kbps video and audio feed to 20 clients. When packed densely to 2 physical nodes, the interpacket gap variance is high, but if we set the estimated bandwidth for the client links to 100 Mb, sparser virtual to physical link mapping results. This in turn forces virtual nodes to relocate onto other physical nodes, raising the total number physical nodes to 6 (see Table 2). The oversubscription of network bandwidth thus clears a path for time sensitive packets.

8 Related Work

The ModelNet network emulator [26] achieves extremely large scale by foregoing flexibility and optionally abstracting away detail in the interior of a network topology. Edge hosts run the user’s applications on generic OSes, using IP aliasing and a socket interposition library to give a weak notion of virtual machine, called a VN. The VNs route their traffic through one or more physical “core” machines that emulate the link characteristics of the interior topology. ModelNet has emulated topologies in excess of 10,000 links. However, it cannot emulate arbitrary computation in the core of a topology, which excludes simple applications like traceroute as well as more complex services like user-configurable dynamic routing, unless support for each feature is hard-wired in (as has been done for DSR) [25].

Compared to Emulab, ModelNet is less transparent to applications and it is harder to provide performance monitoring, because it currently uses only a very weak notion of virtual machine. For example, it does not virtualize filesystem namespace, VNs cannot be multihomed, and it provides no network bandwidth isolation between VNs on the same physical host. ModelNet and the new Emulab are clearly complementary—ModelNet is perfect for generic network interiors, while the new Emulab

is strong in other ways.

Building on ModelNet and the Xen [3] virtual machine monitor, DieCast [11] uses time dilation to run large virtual experiments. In DieCast, time is “slowed down” inside of the virtual machines by an amount equal to the multiplexing factor, resulting in an experiment that takes much longer to execute, but which provides the illusion that the full capacity of the host CPU, network bandwidth, and other “time-scalable” resources are available to each virtual node. As a result of this time dilation, each DieCast virtual node has more of these resources available to it than ours do, but the overall efficiency of the testing facility is not improved. Thus, our virtual nodes are more appropriate for a shared facility. DieCast represents an alternative approach to scale up experimentation resources, bringing with it a different set of challenges to solve.

The Virtual Internet architecture [24] is a partially implemented model targeted to deploying virtual IP networks as overlay networks on the live Internet. The VI work identified most of the issues with link virtualization at the IP layer that we encountered at the Ethernet level. It focuses on correct implementation of virtual links when nodes can simultaneously participate in multiple topologies (concurrency), as multiple nodes in a single topology (revisitation) and when nodes in a virtual topology can themselves act as base nodes for other topologies (recursion). It does not virtualize other node resources.

Virtual machines have a long history, but we discuss only a few recent examples that have been used specifically to implement network emulation environments. This related work generally concentrates on node and/or network virtualization, but we provide a complete system including experimenter control, automated resource assignment and feedback directed virtualization.

IMUNES [32] is an integrated network emulation environment using FreeBSD jail-based virtual nodes and the “vimage” virtual network infrastructure work [30, 31] (which is now part of FreeBSD-CURRENT, but was not available when we started). Rather than virtualize pieces of the network stack, the authors virtualize the entire stack and associate an instance with each jail. While conceptually cleaner, the complete duplication of all network resources raises issues of kernel memory fragmentation. Their implementation provides some basic control over CPU usage that ours currently does not. Although IMUNES topologies can span multiple physical machines, they do not have the automation support to layout and control such topologies.

The node virtualization facility added to the Network Emulation Testbed (NET) [16] provides a lightweight virtual node mechanism in Linux based on virtual routing tables and custom Linux modifications. Their en-

vironment provides wireless as well as wired network emulation. The NET virtual networking implementation is analogous to ours, with their “vnmux” virtual interface and bridge taking the place of our “veth” device and the “NETshaper” replacing our Dummynet usage. Some degree of application transparency is achieved by using *chvrf*, a Linux chroot-like utility, to separate process and network name spaces. The NET work is highly complementary to ours in that it provides a Linux virtual node implementation as well as wireless network emulation that could be integrated with Emulab.

PlanetLab [20] is a geographically distributed network testbed, with machines time-shared among mutually untrusting users. PlanetLab uses Linux vservers [14] enhanced with a custom kernel module that provides enhanced resource isolation, including CPU and network bandwidth. Node virtualization is constrained by the fact that the nodes are subject to the restrictions of the site at which they reside. For example, since they cannot assume more than a single routable IP address is available per node, IP name space is not virtualized.

VINI [4] is a virtual network infrastructure designed to allow multiple, simultaneous experiments with arbitrary network topologies to run on a “real” shared physical network infrastructure. Specifically, PL-VINI is an implementation of VINI on PlanetLab nodes. It builds on top of PlanetLab vservers, adding virtual routers connected by virtual point-to-point links along with the ability to direct real Internet traffic through the resulting virtual network. The absolute performance of PL-VINI was poor due to the need to implement forwarding infrastructure in user mode on the PlanetLab Linux kernel. It also offers only rudimentary traffic shaping and topology setup mechanisms.

A new implementation of VINI called Trellis [5] improves the performance and capabilities of PL-VINI by moving the virtual networking into the Linux kernel, enabling faster packet forwarding and traffic shaping via standard Linux tools. We are currently collaborating with the VINI developers to bring VINI nodes under Emulab control, enabling the full power of Emulab’s experiment creation and control infrastructure.

Auto adaptation, using an automated iterative process to best match a workload to available resources, is also not a new idea. One example is Hippodrome [1], a tool for optimizing storage system configurations. Hippodrome uses storage-relevant metrics (e.g., IOs/sec) to analyze a target workload. It feeds that information into a “solver” which uses modeling to find a good candidate storage architecture, then reconfigures the underlying storage subsystem accordingly. This process is repeated until a configuration is found that satisfies the workload’s IO requirements.

Compared with our work, Hippodrome is focused on

a much narrower set of resources. They are concentrated on IO bandwidth where we must consider a workload's CPU, memory and network resource requirements as well as storage requirements. As a result, they can use more sophisticated and specialized analysis and design tools (e.g., storage system models), allowing quicker convergence on a suitable resource configuration.

9 Discussion and Conclusion

Our resource allocation and monitoring techniques do not assure the *timeliness* of events. In general, assured timeliness is expensive to provide, requiring real-time scheduling of CPU and links. However, we do provide two ways to address the issue, with another planned. First, the user's application-specific metrics, if they can be gathered on unmultiplexed nodes, serve as a safety mechanism to catch arbitrary performance infidelities. Second, the user can specify a shorter time period (the default is 1 second) over which the monitoring daemon will average, as it looks for overload. Finally, we may add a kernel mechanism that will detect if any resource use over very fine time scales, e.g., 1–10 msecs, has exceeded a user-settable threshold. Given this mechanism and typical Internet latencies, users can be quite confident that timing effects regarding network I/O have not affected their experiments.

Evaluation of packet timeliness and CPU scheduling effects remain to be done, but by offering the user application-level metrics directing adaptation, that is not essential. Exhaustive validation of the link emulation fidelity should be done, similar to the inter-packet arrival and time-variance analysis we do for mixed simulated/emulated resources [13]. Another issue is that our default mode of encapsulation decreases the MTU by a few bytes, which could affect some applications. In this case we support two other techniques that require no loss of MTU size: the virtual network devices can be configured to use fake MAC addresses in place of encapsulation, or to use 802.1Q VLAN tagging. We may add well-known OS resource isolation mechanisms such as proportional-share scheduling and resource containers. In a completely different but important area, some aspects of Emulab's Web-based user interface, such as its Graphviz-based topology visualization, are inconvenient to use on thousands of nodes. In response we have built on Munzner's hyperbolic three-dimensional graph explorer library [18] to provide an interactive "fish-eye" visualizer for Emulab, though have not yet put it into production use. Finally, our node support is limited to FreeBSD, yet many want Linux or Windows. When the Trellis work is mature we plan to adopt that to obtain equivalent support for Linux. We currently have Xen partially supported in Emulab, and are exploring

VMware [27].

In conclusion, we have identified, designed, and implemented the many features necessary to support practical scalable network experimentation, and deployed them in a production system. We have shown that, by relaxing the constraints of conservative resource allocation, we can significantly increase the scale of topologies that we can support, or lower the required physical resources, with minimal loss of fidelity. In the future we will gather experience on how experimenters use the feedback and adaptation system, and evolve our system accordingly.

Acknowledgments

Many members of the Flux Research Group assisted with this work. We owe special thanks to Eric Eide for significant writing and \LaTeX assistance, to Russ Fish for the Hypview visualizer, and to Sai Susarla for helping with evaluation.

We also thank the anonymous reviewers and our shepherd, Zheng Zhang, whose comments helped us to improve this paper. This material is based upon work largely supported by NSF grants 0082493, 0205702, and 0335296, and by hardware grants from Cisco Systems.

References

- [1] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proc. of the Conf. on File and Storage Technologies (FAST)*, pages 175–188, Monterey, CA, Jan. 2002.
- [2] Apple Inc. Open Source Streaming Server. <http://developer.apple.com/opensource/server/streaming>.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, Oct. 2003.
- [4] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. of SIGCOMM*, pages 3–14, Pisa, Italy, Sept. 2006.
- [5] S. Bhatia, M. Motiwala, W. Mühlbauer, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford. Hosting Virtual Networks on Commodity Hardware. Technical Report GT-CS-07-10, Department of Computer Science, Georgia Tech, Jan. 2008.
- [6] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulation. 33(5):59–67, May 2000. (An expanded version is available as USC CSD TR 99-702b.).
- [7] C. Dovrolis, P. Ramanathan, and D. Moore. What Do Packet Dispersion Techniques Measure? In *Proc. of IEEE INFOCOM*, pages 905–914, Anchorage, AK, Apr. 2001.
- [8] J. Duerig, R. Ricci, J. Byers, and J. Lepreau. Automatic IP Address Assignment on Network Topologies. Flux Technical Note FTN-2006-02, University of Utah, Feb. 2006. <http://www.cs.utah.edu/flux/papers/ipassign-ftn2006-02.pdf>.
- [9] K. Fall. Network Emulation in the Vint/NS Simulator. In *Proc. of the 4th IEEE Symp. on Computers and Communications (ISCC)*, pages 244–250, Sharm El Sheik, Red Sea, Egypt, July 1999.

- [10] GENI Planning Group. GENI Facility Design. GENI Design Document GDD-07-44, GENI, Mar. 2007. <http://geni.net/GDD/GDD-07-44.pdf>.
- [11] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proc. of NSDI*, pages 407–421, San Francisco, CA, Apr. 2008.
- [12] S. Guruprasad. Issues in Integrated Network Experimentation using Simulation and Emulation. Master's thesis, University of Utah, Aug. 2005. <http://www.cs.utah.edu/flux/papers/guruprasad-thesis-base.html>.
- [13] S. Guruprasad, R. Ricci, and J. Lepreau. Integrated Network Experimentation using Simulation and Emulation. In *Proc. of the First Intl. Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, pages 204–212, Trento, Italy, Feb. 2005.
- [14] Linux-VServer Project. <http://www.linux-vserver.org/>.
- [15] P. Mahadevan, C. Hubble, D. Krioukov, B. Huffaker, and A. Vahdat. Orbis: Rescaling Degree Correlations to Generate Annotated Internet Topologies. In *Proc. of SIGCOMM*, pages 325–336, Kyoto, Japan, Aug. 2007.
- [16] S. Maier, D. Herrscher, and K. Rothermel. On Node Virtualization for Scalable Network Emulation. In *Proc. of the 2005 Intl. Symp. on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 917–928, Philadelphia, PA, July 2005.
- [17] METIS Family of Multilevel Partitioning Algorithms Web Page. <http://www-users.cs.umn.edu/~karypis/metis/>.
- [18] T. Munzner. Exploring Large Graphs in 3D Hyperbolic Space. *IEEE Computer Graphics and Applications*, 18(4):18–23, 1998.
- [19] S. Nanda and T. Chiueh. A Survey on Virtualization Technologies. Technical Report ECSL-TR-179, SUNY at Stony Brook, Feb. 2005.
- [20] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. of HotNets-I*, Princeton, NJ, Oct. 2002.
- [21] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *ACM SIGCOMM Computer Communication Review (CCR)*, 33(2):65–81, Apr. 2003.
- [22] R. Scandariato and F. Risso. Advanced VPN support on FreeBSD systems. In *Proc. of the 2nd European BSD Conf.*, Amsterdam, The Netherlands, Nov. 2002.
- [23] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of SIGCOMM*, pages 133–145, Pittsburgh, PA, Aug. 2002.
- [24] J. D. Touch, Y.-S. Wang, L. Eggert, and G. G. Finn. A Virtual Internet Architecture. Technical Report ISI-TR-2003-570, Information Sciences Institute, Mar. 2003.
- [25] A. Vahdat. Personal communication, May 2004.
- [26] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. of the 5th Symp. on Operating Systems Design and Impl. (OSDI)*, pages 271–284, Boston, MA, Dec. 2002.
- [27] VMware, Inc. VMware: A Virtual Computing Environment. <http://www.vmware.com/>, 2001.
- [28] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the 5th Symp. on Operating Systems Design and Impl. (OSDI)*, pages 255–270, Boston, MA, Dec. 2002.
- [29] J. Winick and S. Jamin. Inet-3.0: Internet Topology Generator. Tech Report CSE-TR-456-02, University of Michigan, 2002.
- [30] M. Zec. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *Proc. of the FREENIX Track: 2003 USENIX Annual Tech. Conf.*, pages 137–150, San Antonio, TX, June 2003.
- [31] M. Zec and M. Mikuc. Real-Time IP Network Simulation at Gigabit Data Rates. In *Proc. of the 7th Intl. Conf. on Telecommunications*, Zagreb, Croatia, June 2003.
- [32] M. Zec and M. Mikuc. Operating System Support for Integrated Network Emulation in IMUNES. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT Infrastructure (OASIS)*, Boston, MA, 2004.
- [33] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of IEEE INFOCOM*, pages 594–602, San Francisco, CA, Mar. 1996.

FlexVol: Flexible, Efficient File Volume Virtualization in WAFL

John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair,
Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini,
Ashish Prakash, Keith A. Smith, Edward Zayas
NetApp, Inc.

Abstract

Virtualization is a well-known method of abstracting physical resources and of separating the manipulation and use of logical resources from their underlying implementation. We have used this technique to virtualize file volumes in the WAFL[®] file system, adding a level of indirection between client-visible volumes and the underlying physical storage. The resulting virtual file volumes, or *FlexVol*[®] volumes, are managed independent of lower storage layers. Multiple volumes can be dynamically created, deleted, resized, and reconfigured within the same physical storage container.

We also exploit this new virtualization layer to provide several powerful new capabilities. We have enhanced SnapMirror[®], a tool for replicating volumes between storage systems, to remap storage allocation during transfer, thus optimizing disk layout for the destination storage system. *FlexClone*[®] volumes provide writable Snapshot[®] copies, using a FlexVol volume backed by a Snapshot copy of a different volume. FlexVol volumes also support thin provisioning; a FlexVol volume can have a logical size that exceeds the available physical storage. FlexClone volumes and thin provisioning are a powerful combination, as they allow the creation of light-weight copies of live data sets while consuming minimal storage resources.

We present the basic architecture of FlexVol volumes, including performance optimizations that decrease the overhead of our new virtualization layer. We also describe the new features enabled by this architecture. Our evaluation of FlexVol performance shows that it incurs only a minor performance degradation compared with traditional, nonvirtualized WAFL volumes. On the industry-standard SPEC SFS benchmark, FlexVol volumes exhibit less than 4% performance overhead, while providing all the benefits of virtualization.

1 Introduction

Conventional file systems, such as FFS [18], ext2 [3], or NTFS [9], are allocated on dedicated storage. Each file

system, representing a single namespace tree, has exclusive ownership of one or more disks, partitions, and/or RAID groups, which provide the underlying persistent storage for the file system. The controlling file system is solely responsible for all decisions about the allocation and use of individual storage blocks on these devices.

With the continuing growth in disk capacities, this has become an increasingly inefficient way to manage physical storage. Larger storage capacities generate a commensurate pressure to create larger file systems on larger RAID groups. This optimizes for performance and efficient capacity utilization. A large number of spindles provides good performance; combining many disks into one large file system makes it easy to dynamically allocate or migrate storage capacity between the users or applications sharing the storage. But this growth in storage capacity can be challenging for end users and administrators, who often prefer to manage their data in logical units determined by the size and characteristics of their application datasets.

As a result, administrators have faced competing pressures in managing their storage systems: either create large file systems to optimize performance and utilization, or create smaller file systems to facilitate the independent management of different datasets. A number of systems, ranging from the Andrew file system [13, 23] to ZFS [26], have addressed these competing needs by allowing multiple file systems, or namespace trees, to share the same storage resources. By separating the management of file systems from the management of physical storage resources, these systems make it easier to create, destroy, and resize file systems, as these operations can be performed independent of the underlying storage.

NetApp[®] has followed a similar evolution with its WAFL file system [11]. For most of its history, users have allocated WAFL file systems (or *volumes* in NetApp terminology) on dedicated sets of disks configured as one or more RAID groups. As a result, WAFL presented the same management challenges as many other file systems. Customers who combined separate file sets on a single volume were forced to manage these files as

a single unit. For example, WAFL Snapshot copies operate at the volume level, so an administrator had to create a single Snapshot schedule sufficient to meet the needs of all applications and users sharing a volume.

In this paper we describe *flexible volumes*, a storage virtualization technology that NetApp introduced in 2004 in release 7.0 of Data ONTAP®. By implementing a level of indirection between physical storage containers (called *aggregates*) and logical volumes (*FlexVol volumes*), we virtualize the allocation of volumes on physical storage, allowing multiple, independently managed file volumes, along with their Snapshot copies, to share the same storage.

Adding a level of indirection allows administrators to manage datasets at the granularity of their choice. It also provides a mechanism for seamlessly introducing new functionality. We have used this indirection to implement writable Snapshot copies (called *FlexClone volumes*), thin provisioning, and efficient remote mirroring.

As is often the case, introducing a new level of indirection brings both benefits and challenges. Mapping between virtual block addresses used by FlexVols and physical block addresses used by aggregates can require extra disk I/Os. This could have a significant impact on I/O-intensive workloads. We have introduced two important optimizations that reduce this overhead. First, dual VBN mappings cache physical block numbers in the metadata for a FlexVol volume, eliminating the need to look up these mappings on most I/O requests. Second, lazily reclaiming freed space from a FlexVol volume dramatically reduces the I/O required to update the virtual-to-physical block mappings. With these optimizations we find that the performance of the SPEC SFS benchmark when using a FlexVol volume is within 4% of the performance of a traditional, nonvirtualized WAFL volume.

In the remainder of this paper, we first describe the design and implementation of flexible volumes, including an overview of the WAFL file system and a description of some of the new functionality enabled by FlexVol volumes. Next we describe our tests comparing FlexVol performance to that of traditional WAFL volumes. Finally, we survey other systems with similar goals and ideas and present our conclusions.

2 Background

This section provides a brief overview of the WAFL file system, the core component of NetApp's operating system, Data ONTAP. The original WAFL paper [11] provides a more complete overview of WAFL, although some details have changed since its publication. In this section, we focus on the aspects of WAFL that are most

important for understanding the FlexVol architecture.

Throughout this paper we distinguish between file systems and volumes. A file system is the code and data structures that implement a persistent hierarchical namespace of files and directories. A volume is an instantiation of the file system. Administrators create and manage volumes; users store files on volumes. The WAFL file system implements these volumes.

WAFL uses many of the same basic data structures as traditional UNIX® style file systems such as FFS [18] or ext2 [3]. Each file is described by an *inode*, which contains per-file metadata and pointers to data or indirect blocks. For small files, the inode points directly to the data blocks. For large files, the inode points to trees of indirect blocks. In WAFL, we call the tree of indirect blocks for a file its *bufree*.

Unlike FFS and its relatives, WAFL's metadata is stored in various metadata files. All of the inodes in the file system are stored in the *inode file*, and the block allocation bitmap is stored in the *block map file*.

These data structures form a tree, rooted in the *vol_info* block. The *vol_info* block is analogous to the superblock of other file systems. It contains the inode describing the inode file, which in turn contains the inodes for all of the other files in the file system, including the other metadata files.

WAFL can find any piece of data or metadata by traversing the tree rooted at the *vol_info* block. As long as it can find the *vol_info* block, it doesn't matter where any of the other blocks are allocated on disk. This leads to the eponymous characteristic of WAFL—its *Write Anywhere File Layout*.

When writing a block to disk (data or metadata), WAFL never overwrites the current version of that block. Instead, the new value of each block is written to an unused location on disk. Thus, each time WAFL writes a block, it must also update any block that points to the old location of the block (which could be the *vol_info* block, an inode, or an indirect block). These updates recursively create a chain of block updates that reaches all the way up to the *vol_info* block.

If WAFL performed all of these updates for each data write, the extra I/O activity would be crippling. Instead, WAFL collects many block updates and writes them to disk *en masse*. This allows WAFL to allocate a large number of blocks to a single region in RAID, providing good write performance. In addition, many of the written blocks are typically referenced from the same indirect blocks, significantly reducing the cost of updating the metadata tree. Each of these write episodes completes when the *vol_info* block is updated, atomically advancing the on-disk file system state from the tree rooted at the old *vol_info* block to the tree rooted at the new one. For this reason, each of these write episodes is called a

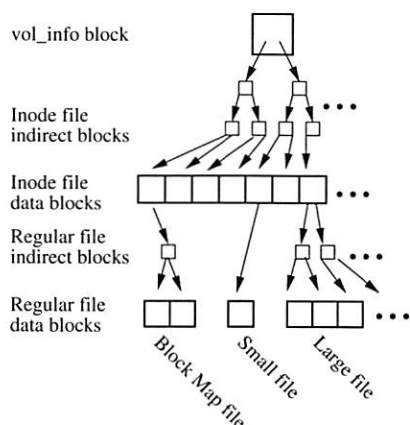


Figure 1: WAFL data structures

consistency point or *CP* for short.

WAFL uses non-volatile memory to log all incoming requests. This ensures that no data is lost in the event of a failure and allows WAFL to acknowledge write requests as soon as they are logged, rather than waiting until the next CP has completed and the data is on disk. After a failure, WAFL returns to the most recently committed CP and replays the contents of the NVRAM log, similar to error recovery in a journaling file system [22].

One of the benefits of the WAFL write anywhere allocation scheme is that it can create point-in-time Snapshot copies of a volume almost for free. Each CP results in a completely consistent on-disk file system image rooted at the current vol_info block. By preserving an old vol_info block and the tree rooted from it, we can create a Snapshot copy of the file system at that point in time. These Snapshot copies are space efficient. The only differences between a Snapshot copy and the live file system are the blocks that have been modified since the Snapshot copy was created (and the metadata that points to them). In essence, WAFL implements copy-on-write as a side effect of its normal operation.

3 FlexVol Architecture

Prior to the introduction of FlexVol volumes, Data ON-TAP statically allocated WAFL volumes to one or more RAID groups. Each disk and each RAID group would belong exclusively to a single volume. We call this style of configuration, which is still supported in Data ON-TAP today, a *traditional volume*, or a *TradVol*.

Our goal, in creating FlexVol volumes, was to break this tight bond between volumes and their underlying storage. Conceptually, we wanted to aggregate many disks into a large storage container and allow administrators to create volumes by carving out arbitrarily sized logical chunks of this storage.

Thinking about this problem, we realized the re-

lationship between volumes and physical storage is the same as that between files and a volume. Since we had a perfectly good file system available in WAFL, we used it to implement FlexVol volumes. This is the essence of the architecture: a FlexVol volume is a file system created *within* a file on an underlying file system. A hidden file system spans a pool of storage, and we create externally visible volumes inside files on this file system. This introduces a level of indirection, or virtualization, between the logical storage space used by a volume and the physical storage space provided by the RAID subsystem.

3.1 Aggregates

An aggregate consists of one or more RAID groups. This storage space is organized as a simple file system structure that keeps track of and manages individual FlexVol volumes. It includes a bitmap indicating which blocks in the aggregate are allocated. The aggregate also contains a directory for each FlexVol volume. This directory serves as a repository for the FlexVol volume and associated data, and it provides a uniform repository for volume-related metadata. It contains two important files for each FlexVol volume, the *RAID file* and the *container file*.

The RAID file contains a variety of metadata describing the FlexVol volume, including its volume name, file system identifier, current volume state, volume size, and a small collection of other information. We call this the RAID file because the corresponding information for a TradVol is stored as part of the RAID label (along with RAID configuration information).

The container file contains all the blocks of the FlexVol volume. Thus, the block addresses used within a FlexVol volume refer to block offsets within its container file. In some respects, the container file serves as a virtual disk containing the FlexVol volume, with significant differences as described in later sections.

Since the container file contains every block within a FlexVol volume, there are two ways to refer to the location of a given block. The physical volume block number (PVCN) specifies the block's location within the aggregate. This address can be used to read or write the block to RAID. The virtual volume block number (VVCN) specifies the block's offset within the container file.

To better understand VVCNs and PVCNs, consider the process of finding the physical disk block given an offset into a file within a FlexVol volume. WAFL uses the file's buftree to translate the file offset to a VVCN—a block address within the FlexVol volume's virtual block address space. This is the same as the way a traditional file system would translate a file offset to a disk

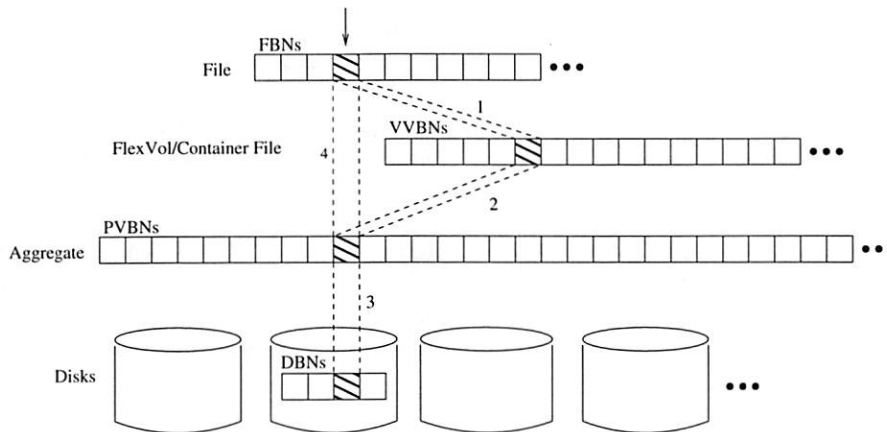


Figure 2: Mapping a block from file to disk. We show a single block as part of several logical and physical storage containers—a file, a container file holding a FlexVol volume, an aggregate, and a disk. Each provides an array of blocks indexed by the appropriate type of block number. The file is indexed by file block number (FBN), the container file by VVBN, and the aggregate by PVBN. Finally, the disks are indexed by disk block number (DBN). To translate an FBN to a disk block, WAFL goes through several steps. In step 1, WAFL uses the file's inode and buftree to translate the FBN to a VVBN. In step 2, WAFL translates the VVBN to a PVBN using the container file's inode and buftree. Finally, in step 3, RAID translates the PVBN to a DBN. Step 4 shows the short-cut provided by dual VBNs (see Section 3.3). By storing PVBNs in the file's buftree, WAFL bypasses the container map's VVBN-to-PVBN translation.

address. The FlexVol volume's block address space is defined by the container file. WAFL uses the container file's buftree to translate the VVBN to a block address in the aggregate's block address space. This provides a PVBN, which WAFL can give to the RAID subsystem to store or retrieve the block. Figure 2 displays this mapping process and the relationship between the different types of block numbers.

Observe that in the container file's buftree, the first level of indirect blocks list all of the PVBNs for the container file. Together, these blocks form an array of PVBNs indexed by VVBN. We refer to the VVBN-to-PVBN mapping provided by this first level of indirect data in the container file as the *container map*.

3.2 Volumes

Because FlexVol volumes are implemented by container files, they inherit many characteristics of regular files. This provides management flexibility, which we can expose to users and administrators.

When a FlexVol volume is created, its container file is sparsely populated; most of the logical offsets have no underlying physical storage. WAFL allocates physical storage to the container file as the FlexVol volume writes data to new logical offsets. This occurs in much the same way as hole-filling of a sparse file in a conventional file system. This sparse allocation of container files also allows the implementation of thin provisioning, as described in Section 4.3.

The contents of a FlexVol volume are similar to those of a traditional WAFL volume. As with a TradVol,

there is a `vol_info` block, located at well-known locations within the container file space. Within the volume are all of the standard WAFL metadata files found in a TradVol.

A FlexVol volume contains the same block allocation files as a traditional volume. While the aggregate-level versions of these files are indexed by PVBN, in a FlexVol volume these files are indexed by VVBN. Thus, the files and the associated processing scale with the logical size of the volume, not with the physical size of the aggregate.

3.3 Dual Block Numbers

The use of VVBNs introduces a layer of indirection which, if not addressed, may have significant impact on read latencies. In a naïve implementation, translating a file offset to a physical block (PVBN) would require WAFL to read two buftrees—one for the file to find the VVBN and one for the container file to find the PVBN. In the worst case, this overhead could be quite high, as WAFL might have to perform this translation for each indirect block as it traverses the file's buftree.

To address this problem, FlexVol volumes use *dual VBNs*. Each block pointer in a FlexVol volume contains two block addresses, the VVBN and its PVBN translation. For normal read operations, WAFL never needs to look up PVBNs in the container file buftree. It just uses the PVBN values it finds in the dual VBNs stored in the inodes and indirect blocks of the FlexVol volume. Figure 3 illustrates this process.

For writes, WAFL allocates a VVBN from the FlexVol volume's container file and a PVBN from its ag-

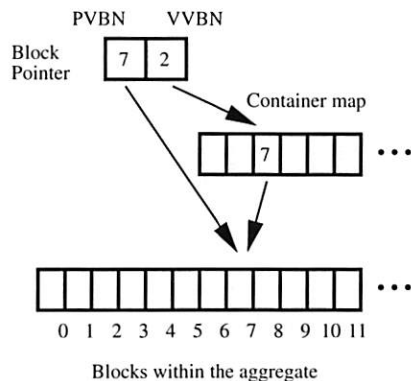


Figure 3: Dual volume block numbers in indirect blocks. A block pointer within a FlexVol volume contains two block addresses. One is the FlexVol volume's VVBN, the location of the block within the flexible volume's container file. The other is the aggregate's PVBN, the physical location of the block. In this example the read path bypasses the container map and goes directly to PVBN #7. The VVBN of 2 is the index into the container map that can be used as an alternate means to find the PVBN.

gregate, and it updates the file and container file buftrees accordingly. This requires extra processing and I/O. Because WAFL delays write allocation until it writes a consistency point, this work occurs asynchronously and does not add latency to the original write request. In workloads with good locality of reference, the overhead is further reduced by amortizing it across multiple updates to the same buftrees.

3.4 Delayed Block Freeing

When a block is freed—for example, by a file deletion—we would like to mark it as free in the aggregate as well as in the FlexVol volume. In other words, we want FlexVol volumes to return their free space to the underlying aggregate. Thus, when a block is no longer referenced from the active file system or from any Snapshot copies we not only mark it free in the FlexVol volume's block map, we also free the block in the aggregate's block map and mark the corresponding location in the container file as unallocated, effectively “punching a hole” in the container file.

The fact that unused blocks are eventually returned from the FlexVol volume to the aggregate is a key feature of the FlexVol design. Without this functionality, freed blocks would remain allocated to their FlexVol. The aggregate would not know that these blocks were free and would not be able to allocate them to other volumes, resulting in artificial free space fragmentation. In contrast, in our implementation, free space is held by the aggregate, not the FlexVol volume; the free space in the aggregate can be made available to any volume within the ag-

gregate. Most importantly, the unrestricted flow of free space requires no external intervention and no management.

This mechanism also motivates an important performance optimization for freeing blocks. Since WAFL always writes modified data to new locations on disk, random overwrites on a large file tend to produce random frees within the VVBN space of the FlexVol volume. This results in random updates of the container file's indirect blocks, adding an unacceptable overhead to random updates of large files.

WAFL avoids this problem by delaying frees from the container file in order to batch updates. WAFL maintains a count of the number of *delayed free* blocks on each page of the container map. Up to two percent of a FlexVol volume's VVBN space can be in the delayed free state. Once the number of delayed free blocks crosses a one percent threshold, newly generated delayed frees trigger background cleaning. The cleaning of the container file is focused on regions of the container block file that have larger than average concentrations of delayed free blocks. Since an indirect block in the container file buftree has 1,024 entries, an average indirect block of the container will hold at least ten (1% of 1,024) delayed frees, and often significantly more. This reduces the container file overhead of freeing blocks to less than one update per ten frees.

4 New Features

Adding a level of indirection between the block addressing used by a FlexVol volume and the physical block addressing used by the underlying RAID system creates a leverage point that we have used to introduce new functionality and optimizations. In this section, we describe three such improvements—volume mirroring for remote replication, volume cloning, and thin provisioning.

4.1 Volume Mirroring

Volume SnapMirror[20] is a replication technology that mirrors a volume from one system to another. SnapMirror examines and compares block allocation bitmaps from different Snapshot copies on the source volume to determine the set of blocks that it must transfer to transmit the corresponding Snapshot copy to the remote volume. This allows SnapMirror to efficiently update the remote volume by advancing its state from Snapshot copy to Snapshot copy.

Since the decisions of which blocks to transfer and where to place them at the destination are based on the allocation maps, they are based on the type of VBN that serves as the index to those files. In a traditional vol-

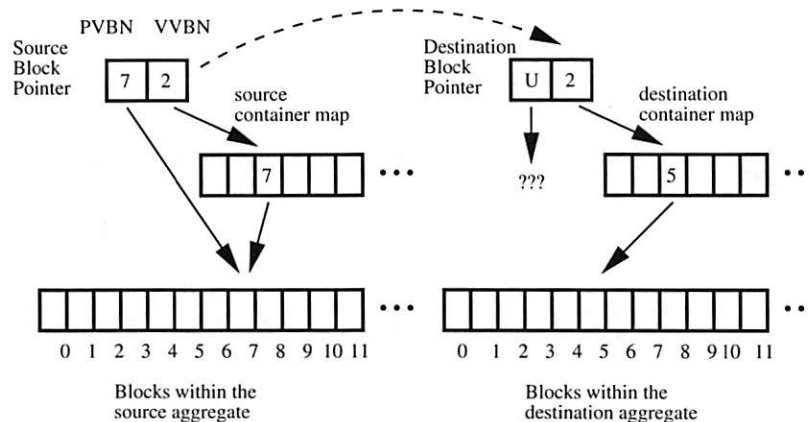


Figure 4: SnapMirror Transfer. *SnapMirror* transfers blocks from a source volume to a destination volume based on VVBNs. The transfers are independent of the physical block numbers involved. Here we show how *SnapMirror* updates a block pointer as a block is transferred. On the source, the block has a VVBN of 2 and a PVBN of 7. When the block is transferred, the destination system assigns it a new PVBN (5) and enters that in the destination FlexVol volume's container map. When it copies the block pointer, *SnapMirror* preserves the VVBN; the block has the same logical address within both FlexVols. The PVBN of the destination block pointer is set to PVBN-UNKNOWN (indicated by a 'U' above). A background process will eventually replace this with the correct PVBN. If WAFL needs to resolve the block pointer before the PVBN is filled in, it can look up the PVBN using the container map.

ume, file block pointers are PVBNs. Thus, to transfer a block, WAFL reads it from RAID using the PVBN and transfers it to the destination where it is written to the same PVBN. If the physical geometries of the source and destination differ, WAFL cannot optimize the I/O for the transfer on both the source and the destination. This is particularly problematic when transferring blocks between systems with drives that have different sizes and geometries, such as when transferring between smaller, faster primary storage disks and larger, slower secondary storage disks.

In contrast, flexible volume transfers are VVBN-based. WAFL uses the FlexVol volume's block allocation files to determine which VVBNs to transfer, and it transfers those blocks from the container file on the source system to the container file at the destination. The destination system assigns a new PVBN to each block while maintaining the same VVBN as on the source system. This removes geometry restrictions from Volume SnapMirror because the source and destination make physical allocation decisions independently. As a result, volumes can be mirrored between aggregates with different sizes and/or disk configurations.

Changing the PVBNs of the volume across a transfer introduces a difficulty. Among the blocks transferred are metadata blocks that contain block pointers, particularly inode file blocks and indirect blocks. Since the VVBN-to-PVBN mapping at the destination is different from that at the source, all of the PVBNs in block pointers must be changed as part of the transfer. To allow SnapMirror to locate the block pointers within blocks, WAFL maintains a block type file that

identifies the general function of each block within the volume, indexed by VVBN. As part of the transfer itself, PVBNs are replaced with a special reserved value, *PVBN-UNKNOWN*. The destination must then replace the *PVBN-UNKNOWN*s with actual PVBNs. Figure 4 illustrates the process of mirroring a single block from one FlexVol volume to another.

Even in the presence of *PVBN-UNKNOWN*, access and use of the destination volume are possible. Any code encountering an unknown PVBN while attempting to read data can instead use the VVBN and the container map to find the PVBN for the required block. This allows access to transferred data while the PVBNs are still being repaired via a background process.

4.2 Volume Cloning

WAFL Snapshot copies provide consistent point-in-time copies of a volume. This has many uses, but sometimes the read-only nature of a Snapshot copy is a limitation. In database environments, for example, it is often desirable to make writable copies of a production database for development or test purposes. Other uses for writable Snapshots copies include upgrades or modifications to large applications and provisioning many systems in a grid or virtual machine environment from a single master disk image.

Volume cloning creates a FlexVol volume in which the active file system is a logical replica of a Snapshot copy in a different FlexVol volume within the same aggregate. The parent volume can be any FlexVol volume, including a read-only mirror. Like the creation of a snap-

shot, creating a flexible volume clone, or *FlexClone volume*, requires the writing of a fixed, small number of blocks and is nearly instantaneous. The FlexClone volume is a full-fledged FlexVol volume with all the features and capabilities of a normal WAFL volume.

Creating a clone volume is a simple process. WAFL creates the files required for a new FlexVol volume. But, rather than creating and writing a new file system inside the volume, WAFL seeds the container file of the clone with a vol_info block that is a copy of the vol_info block of the Snapshot copy on which the clone is based. Since that vol_info is the top-level block in the tree of blocks that form the Snapshot copy, the clone inherits pointers to the complete file system image stored in the original Snapshot copy. Since the clone does not actually own the blocks it has inherited from the parent, it does not have those blocks in its container file. Rather the clone's container has holes, represented by zeros in the container map, indicating that the block at that VVBN is inherited from the parent volume (or some more distant ancestor, if the parent volume is also a clone).

To protect the cloned blocks from being freed and overwritten, the system needs to ensure that the original volume will not free the blocks used by the clone. The system records that the clone volume is relying on the Snapshot copy in the parent volume and prevents the deletion of the Snapshot copy in the parent volume or the destruction of the parent volume. Similarly, WAFL ensures that the clone will not free blocks in the aggregate that are owned by the parent volume. Inherited blocks are easily identified at the time they are freed because the container file of the clone has a hole at that VVBN.

A clone volume can be split from the parent volume. To do this a background thread creates new copies of any blocks that are shared with the parent. As a result of this process, the clone and parent will no longer share any blocks, severing the connection between them.

4.3 Thin Provisioning

As described earlier, the free space within a WAFL aggregate is held by the aggregate. Since FlexVol volumes do not consume physical space for unallocated blocks in their address space, it is natural to consider thin provisioning of volumes. For example, several 1TB volumes can be contained in a 1TB aggregate if the total physical space used by the volumes is less than 1TB. The ability to present sparsely filled volumes of requested sizes without committing underlying physical storage is a powerful planning tool for administrators. Volume clones also present a natural case for thin provisioning, since users will often want many clones of a given volume, but the administrator knows that the clones will all

share most of their blocks with the base snapshot.

While thin provisioning of volumes presents many opportunities to administrators, it also provides challenges. An aggregate can contain many volumes, and no single provisioning policy will suit all of them, so WAFL allows different volumes in the same aggregate to use different policies. The policies are *volume*, *none*, and *file*, which provide a range of options for managing thin provisioning. The three policies differ in their treatment of volume free space and their treatment of space-reserved objects within volumes.

The *volume* policy is equivalent to a space reservation at the aggregate level. It ensures that no other volumes can encroach on the free space of the volume. The *none* policy implements thin provisioning for the entire volume. No space is reserved for the aggregate beyond that currently consumed by its allocated blocks.

The third policy, *file*, exists for cases where writes to specific files should not fail due to lack of space. This typically occurs when clients access a file using a block protocol such as iSCSI. In such a case, the file appears to the client as logical disk device. For such objects, WAFL provides the ability to reserve space for the underlying files. On a volume with a *file* policy, the reservations on individual objects within the volume are honored at the aggregate level. Thus, if a 400GB database table is created on a 1TB FlexVol volume with a *file* policy, the aggregate would reserve 400GB of space to back the database file, but the remaining 600GB would be thinly provisioned, with no storage reservation.

The default policy for a FlexVol volume is *volume*, since this means that volumes behave precisely the same as a fully provisioned TradVol. By default, a FlexClone volume inherits the storage policy of its parent volume.

4.4 Other Enhancements

Over time we have continued to find this new level of virtualization valuable for cleanly implementing new features in WAFL. In addition to the features described above, we are also using this technique to introduce block-level storage deduplication and background defragmentation of files and free space.

5 Evaluation

In this section, we evaluate the performance overhead imposed by the FlexVol architecture. While FlexVol virtualization is not free, we find that the extra overhead it imposes is quite modest. We compare the performance of FlexVol volumes and TradVol volumes using both microbenchmarks and a large-scale workload. We also discuss how customers use FlexVol volumes in production

environments, presenting data drawn from NetApp's installed base. Finally, we comment on our practical experiences with the engineering challenges of introducing FlexVol volumes into Data ONTAP.

5.1 FlexVol Overhead

We compare the performance of FlexVol volumes and TradVol volumes for basic file serving operations and for larger workloads. Our goal is to quantify and understand the overheads imposed by the extra level of indirection that allows us to implement the features of FlexVol volumes.

Intuitively, we expect FlexVol volumes to impose overhead due to the increased metadata footprint required to maintain two levels of buftree. Larger metadata working sets will increase cache pressure, I/O load, and disk utilization. In addition, generating and traversing this increased metadata will add CPU overhead.

To determine where these overheads occur and quantify them, we ran a set of simple micro-benchmarks on both FlexVol volumes and TradVol volumes. By examining the performance differences between the two volume types in conjunction with performance counter data from Data ONTAP, we can measure the specific performance cost of FlexVol volumes.

Finally, to understand how these detailed performance differences affect large-scale benchmarks, we used the industry-standard SPEC SFS benchmark [24] to compare the macro-level performance of both volume types.

5.1.1 Microbenchmarks

For our microbenchmarks, we used an FAS980 server running Data ONTAP version 7.2.2. This system has two 2.8GHz Intel® Xeon processors, 8GB of RAM, and 512MB of NVRAM. The storage consists of 28 disk drives (72GB Seagate Cheetah 10K RPM) configured as two 11 disk RAID-DP® [7] aggregates. One aggregate was used for FlexVol volumes and the other as a TradVol volume. The remaining disks were not used in these tests; they held the root volume or served as spares.

The benchmark we used is called *filersio*. It is a simple tool that runs on a file server as part of Data ONTAP. We used it to generate read and write workloads with either sequential or random access patterns. Because *filersio* runs on the file server, it issues requests directly to WAFL, bypassing the network and protocol stacks. This allows us to focus our investigation on the file system, ignoring the interactions of client caches and protocol implementations.

We used *filersio* to examine four workloads—random reads, sequential reads, random writes, and se-

quential writes. To examine the impact of FlexVols volumes' larger metadata foot print, we ran each test on increasing dataset sizes, ranging from 512MB to 32GB. At 512MB, all of the test data and metadata fits in the buffer cache or our test system. With a 32GB data set, the buffer cache holds less than 20% of the test data. In each test, our data set was a single large file of the appropriate size.

For each test, we ran the workload for 5 minutes. We configured *filersio* to maintain ten outstanding I/O requests. We warm the WAFL buffer cache for these tests by first creating the test file and then performing five minutes of the test workload before we execute the measured test. The I/O request size and alignment for all tests is 4KB, matching the underlying WAFL block size. All of the results are averages of multiple runs, each using a different test file.

Figures 5, 6, 7, and 8 show the results of our microbenchmarks. We note that across most of these benchmarks the FlexVol volume has performance nearly identical to the TradVol. There are three significant areas where FlexVol performance lags—for file sizes less than about 6GB in the random read tests, for file sizes greater than 8GB in the sequential write test, and for all file sizes in the random write test. The largest performance differences were for 32GB random writes, where the FlexVol performed 14% worse than the TradVol. In the remainder of this section, we discuss these differences and also provide some intuition about the general shapes of the performance curves we see in these microbenchmarks.

Figure 5 shows the results of the random read benchmark. This graph can be divided into two regions. For file sizes larger than about 5.8GB, the working set size exceeds the available cache. Performance drops off rapidly in this region, as more and more requests require disk I/O. It is in this region that we might expect to see the performance overhead of the FlexVol volume, as it needs to access additional metadata, both on disk and in memory, to satisfy cache misses. The FlexVol requires twice as many indirect blocks to describe a file of a given size. These extra blocks compete with data blocks for cache space and reduce our hit rate. In fact, the performance of the two configurations is nearly identical, as the increase in cache pressure is negligible. Roughly 1 in 500 cache blocks is used for indirect metadata on a FlexVol volume, compared to 1 in 1000 on a TradVol.

On the left-hand side of Figure 5 our working set fits entirely in the buffer cache. Cache hits in WAFL follow the same code path, regardless of the volume type, and do not read any buftree metadata. Thus, we would expect all tests in this range to perform identically, regardless of file size or volume type. Surprisingly, this was not the case. The performance of both volume types steadily declines as the file size increases, and FlexVol volumes consistently underperform TradVols.

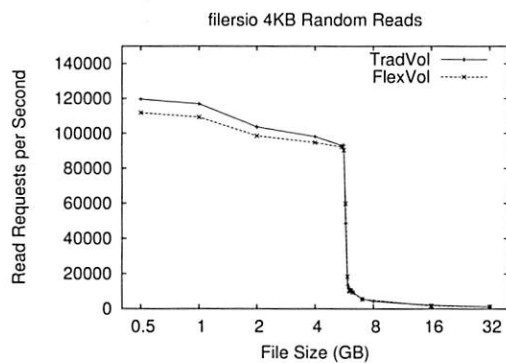


Figure 5: The performance of a random read workload on files of increasing size. Each data point is the average of eight test runs. Standard deviations are less than 5% of the average on all data points except where the working set starts to exceed the size of the buffer cache (5.7–6.4GB). At these sizes standard deviations ranged from 5–30%, reflecting the sensitivity of these tests. With a 100x performance difference between cache hits and cache misses, small differences in cache hit rates, caused by the random number generator, result in large differences in average performance.

To better understand this behavior, we examined the performance statistics Data ONTAP collects. As expected, these tests had 100% hit rates in the buffer cache. Further study of these statistics uncovered the explanations for these behaviors. First, because we have 8GB of RAM on a system with a 32-bit architecture, not all cache hits have the same cost. When WAFL finds a page in the cache that is not mapped into its address space, Data ONTAP must remap the page before returning a data pointer to the page. This accounts for the declining performance as the file size increases. With larger working set sizes, a greater percentage of blocks are cached but not in Data ONTAP's address space, and thus a greater percentage of cache hits pay the page remapping penalty. In the tests with 512MB files, the total number of page remappings is less than 0.001% of the number of cache hits. For the 5.5GB tests, this ratio is 75%.

The second anomaly—the fact that TradVol volumes appear to serve cache hits faster than FlexVol volumes—is explained by a cached read optimization. When WAFL detects a sequence of 5,000 or more successive reads that are serviced from the cache, it stops issuing read-ahead requests until there is a cache miss. Unexpectedly, this optimization is not always enabled during these filersio runs. The culprit is various *scanner* threads that WAFL runs in the background. These threads perform a variety of tasks, such as the delayed-free processing described in Section 3.4. Whenever one

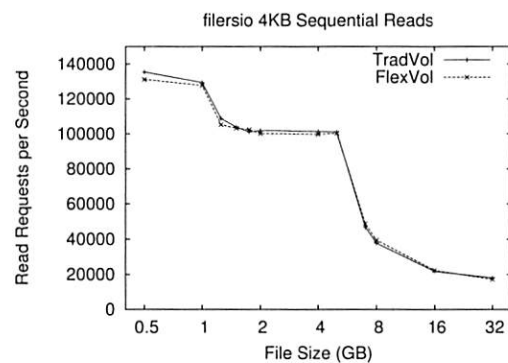


Figure 6: The performance of a sequential read workload on files of increasing size. Each data point is the average of eight runs. Standard deviations are less than 9% of the average at all data points.

of these scanners has a read miss, it disables the cached read optimization on its volume. There is more of this background activity on a FlexVol volume than a TradVol. As a result, the cached read optimization is not enabled as much on the FlexVol volume, causing it to spend extra CPU cycles needlessly looking for blocks to prefetch. Since this workload is completely CPU-bound, this extra overhead has a noticeable effect on performance. In the tests with 512MB files, the cached read optimization was enabled for 82% of the reads from the TradVol, but only for 32% of the reads from the FlexVol volume. Unfortunately, it is not possible to completely disable this background processing in WAFL. It generally causes very little additional I/O, and this was the only test case where it had a noticeable effect on our test results.

Figure 6 shows sequential read performance. The performance on the FlexVol volume is, again, nearly identical to TradVol performance. The overall shape of the performance curves is similar to the random read benchmark with two noteworthy differences. First, the out-of-cache performance is substantially better than the random read case, reflecting the performance gains from prefetching and sequential I/O. The second difference is the step-like performance drop around the 1.5GB file size. This is where the file size exceeds the available mapped buffer cache. For file sizes smaller than this we see almost no page remaps, but for file sizes larger than this the number of page remaps roughly equals the total number of reads during the test.

Figure 7 shows sequential write performance. Here we see similar performance for FlexVol volumes and TradVol volumes at most file sizes. Although FlexVol volumes update more metadata during this benchmark, it has no discernible effect on performance for most file sizes. The sequential nature of the workload provides good locality of reference in the metadata the FlexVol

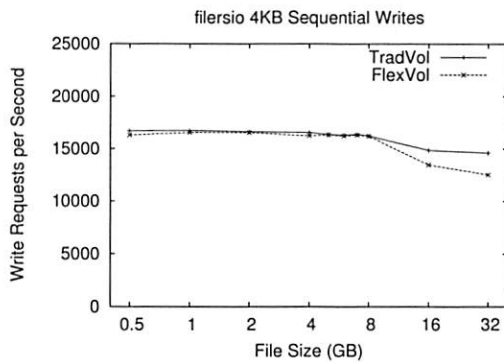


Figure 7: The performance of a sequential write workload on files of increasing size. Each data point is the average of ten runs. Standard deviations are less than 5% of the average at all data points.

volume needs to modify. Thus, the amount of metadata of written to the FlexVol volume is quite modest compared to the file data writes, which dominate this test.

As the file sizes grow past 8GB, sequential write performance on both volumes started to drop, with a larger performance decrease on the FlexVol volume. Our performance statistics are less clear about the cause of this drop. It appears to stem from an increase in the number of cache misses for metadata pages, particularly the allocation bitmaps. This has a larger impact on the FlexVol volume because it has twice as much bitmap information—both a volume-level bitmap and an aggregate-level bitmap.

Finally, in Figure 8 we show random write performance. On both volumes, performance slowly decreased as we increased the file size. This occurs because we have to write more data to disk for larger file sizes. For smaller file sizes, a greater percentage of the random writes in each consistency point are overwrites of a previously written block, reducing the number of I/Os we have to perform in that consistency point. For example, the 512MB FlexVol test wrote an average of 45,570 distinct data blocks per CP; the 32GB FlexVol test wrote an average of 72,514 distinct data blocks per CP.

In the random write test we also see that FlexVol performance lagged TradVol performance. The performance difference ranges from a few percent at the smaller file sizes to 14% for 32GB files. This performance gap reflects several types of extra work that the FlexVol volume performs, most notably the extra I/O associated with updating both the file and container bultrees and the delayed free activity associated with all of the blocks that are overwritten during this test. These same factors occur in the sequential write test, but their performance impact is attenuated by the locality of reference we get in the metadata from doing sequential writes.

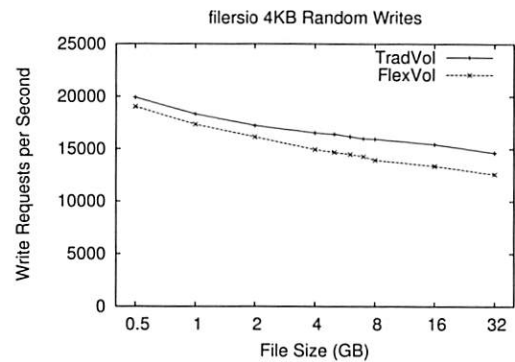


Figure 8: The performance of a random write workload on files of increasing size. Each data point is the average of eight test runs. Standard deviations are less than 8% of the average on all data points.

With random writes, we change much more metadata in each CP, increasing the amount of data that have we to write to disk. The slightly larger performance gap for large file sizes is due to the cache misses loading bitmap files, as we saw in the sequential write test described above.

In summary, most of our microbenchmarks show nearly identical FlexVol and TradVol performance. The major exceptions are cached random reads, random writes, and sequential writes to large files. In these cases, FlexVol performance is often within a few percent of TradVol performance; in the worst cases the performance difference is as much as 14%.

5.1.2 SFS Benchmark

To understand how the behaviors observed in our microbenchmarks combine to affect the performance of a more realistic workload, we now examine the behavior of a large scale benchmark on both FlexVol volumes and TradVol volumes.

For this test, we use the SPEC Server File System (SFS) benchmark [24]. SFS is an industry standard benchmark for evaluating NFS file server performance. SFS originated 1993 when SPEC adopted the LADDIS benchmark [28] as a standard benchmark for NFS servers. We used version SFSv3.0r1, which was released in 2001.

SFS uses multiple clients to generate a stochastic mix of NFS operations from a predefined distribution. The load generating clients attempt to maintain a fixed load level, for example 5,000 operations/second. SFS records the actual performance of the server under test, which may not be the target load level. SFS scales the total data set size and the working set size with the offered load. A complete SFS run consists of multiple runs with

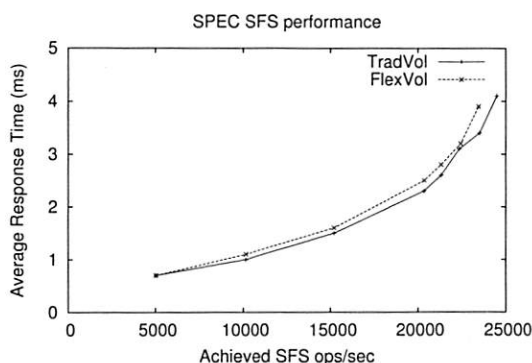


Figure 9: Performance of the SPEC SFS benchmark. This graph plots average response time as a function of the achieved throughput (in operations per second). We performed three SFS runs in each configuration. Here we show the runs with median peak performance. There was little variance across the runs. Throughputs for all load points are within 1.5% of the values shown here. Response times were within 10% of the values shown here, except for the two highest FlexVol load points, where they varied by as much as 18%.

increasing offered load, until the server's performance peaks. At each load point, SFS runs a five minute load to warm the server cache, then a five minute load for measurement. The result from each load point is the achieved performance in operations per second and the average latency per request. Further details about SFS are available from the SPEC web site [24].

Our SFS tests used NFSv3 over TCP/IP. The file server was a single FAS 3050 (two 2.8GHz P4 Xeon processors, 4GB RAM, 512MB NVRAM) with 84 disk drives (72 GB Seagate Cheetah 15K RPM). The disks were configured as a single volume (either flexible or traditional) spanning five RAID-DP [7] groups, each with 14 data disks and 2 parity disks.

Figure 9 shows the results of these tests. As we would expect, given the relatively small performance differences uncovered in our microbenchmarks, the FlexVol and TradVol volumes have similar performance. The TradVol achieves a peak performance of 24,487 ops/sec, 4.4% better than the peak FlexVol performance of 23,452 ops/sec. With increasing loads, there is an increasing gap in response time, with FlexVol volumes showing higher latencies than TradVols. At the peak FlexVol throughput, its average response time of 3.9ms is 15% longer than the corresponding TradVol load point.

Our microbenchmarks exhibited several sources of FlexVol overhead that affect SFS performance, including extra CPU time and I/Os to process and update FlexVol metadata. Performance statistics collected from the file server during the SFS runs show that at comparable load

points the FlexVol volume used 3–5% more CPU than the TradVol, read 4–8% more disk blocks, and wrote 5–10% more disk blocks. A large amount of the extra load came from medium size files (32–64KB) that don't need any indirect blocks on the TradVol but use indirect blocks on the FlexVol volume because of the extra space used by dual VBNs.

Overall, we are pleased with the modest overhead imposed by FlexVol virtualization. FlexVol performance is seldom more than a few percent worse than TradVol performance. In comparison, this overhead is far less than the performance increase seen with each new generation of server hardware. In exchange for this slight performance penalty, customers have increased flexibility in how they manage, provision, and use their data.

5.2 Customer Usage of FlexVol Volumes

NetApp storage systems have a built-in, low-overhead facility for reporting important system events and configuration data back to the NetApp AutoSupport database. The use of this reporting tool is optional, but a large percentage of NetApp customers enable it. Previous studies have used this data to analyze storage subsystem failures [14], latent sector errors [1], and data corruptions [2]. In this section we examine system configuration data to understand how our customers use FlexVol volumes.

We examined customer configuration data over a span of a year from September 2006 to August 2007. At the beginning of this period, our database had information about 38,800 systems with a total of 117.8PB of storage. At the end of this period, we had data from 50,800 systems with 320.3PB of storage. Examining this data, we find evidence that customers have embraced FlexVol technology and rely on it heavily. Over this year, we found that the percentage of deployed systems that use only TradVols decreased from 46% of all systems to less than 30% of all systems. At the same time, the percentage of systems that use only FlexVol volumes increased from 42% to almost 60%. The remainder, systems that had a mixture of TradVol and FlexVol volumes, was fairly stable at 11.5% of the systems in our dataset.

Looking at numbers of volumes instead of systems we found that over the same year the number of FlexVol volumes increased from 69% of all volumes to 84%, while traditional volumes decreased from 31% to 16%.

FlexVol volumes allow customers to manage their data (volumes) at a different granularity than their storage (aggregates). As disk capacities have grown, we see increasing evidence that our customers are using FlexVol volumes in this way—allocating multiple FlexVol volumes on a single aggregate. During the year, the average size of the aggregates in our data set increased from 1.5TB to 2.3TB. At the same time, the average number

of volumes per aggregate increased from 1.36 to 1.96. Excluding traditional volumes, which occupy the whole aggregate, the average number of volumes per aggregate increased from 1.95 to 3.40.

Finally, we explored customer adoption of thinly provisioned FlexVol and FlexClone volumes. Over the past year, the number of thinly provisioned volumes (i.e., with a provisioning policy of *none*) has grown by 128%, representing 6.0% of all FlexVol volumes in the most recent data. The total size of all thin provisioned volumes currently exceeds the physical space available to them by 44%, up from 35% a year ago.

FlexClone volumes represent a smaller fraction of volumes, but their use has been growing rapidly, from 0.05% to 1.0% of all FlexVol volumes during our year of data. Customers are making heavy use of thin provisioning with their clone volumes. In the current data, 82% of all clones are thinly provisioned. We expect that this data under-represents customer adoption of FlexClone volumes, since many use cases involve short-lived clones created for testing and development purposes.

5.3 Experience

While FlexVol volumes provide good performance and have been readily adopted by NetApp customers, the experience of introducing a major piece of new functionality into an existing operating system has not been perfect. The majority of the challenges we faced have come from legacy parts of Data ONTAP, which assumed there would be a modest number of volumes.

One of the major areas where these problems have manifested is in limits on the total number of volumes allowed on a single system. At boot time and at failover, Data ONTAP serially mounts every volume on a system before accepting client requests. With many hundreds or thousands of volumes, this can have an adverse effect on system availability. Likewise, when Data ONTAP supported only a small number of volumes, there was little pressure to limit the memory footprint of per-volume data structures, and no provision was made to swap out volume-related metadata for inactive volumes.

Over time, these constraints have gradually been addressed. Data ONTAP currently restricts the total number of volumes on a single system to 500. This limits failover and reboot times and prevents volume metadata from consuming too much RAM.

6 Related Work

FlexVol is not the first system to allow multiple logical storage containers to share the same physical storage pool. In this section we survey other systems that have

provided similar functionality. We first discuss other systems that provide virtual file systems, contrasting them with FlexVol volumes in terms of implementation and functionality. In the remainder of this section we contrast FlexVol volumes with various systems that provide other virtualized storage abstractions.

The earliest example of a file system supporting multiple file systems in the same storage was the Andrew File System (AFS) [12, 13, 15]. In AFS, each separate file system was called a volume [23]. AFS is a client-server system; on a file server, many volumes are housed on a single disk partition. Clients address each volume independently. Volumes grow or shrink depending on how much data is stored in them, growing by allocating free space within the disk region and shrinking by freeing space for use by others. An administrative limit (or *quota*) on this growth can be set independent of the disk region size. Thus, administrators can implement thin provisioning by overcommitting the free space in the region. AFS maintains a read-only copy-on-write point-in-time image of an active volume, called a *clone* in AFS terminology; the clone shares some of the storage with the active volume, similar to WAFL Snapshots copies. In AFS, clones are always read-only; the writable clones we describe do not exist in AFS.

Howard *et al.* [13] describe an evolution of AFS similar to that of WAFL and FlexVol volumes—moving from a prototype implementation that supported a single file volume per storage device (essentially unmodified 4.2BSD) to a revised system with the multiple volume per container architecture described above. Both the revised AFS implementation as well as FlexVol volumes implement virtualization by providing a layer of indirection: in the AFS case, per-volume inode tables, and in the FlexVols case, per-volume block maps.

Other file systems have since used an architecture similar to volumes in AFS. Coda [21] is a direct descendant of AFS that supports disconnected and mobile clients. DCE/DFS [6, 16], like WAFL, can also accommodate growth of the underlying disk media, for instance by adding a disk to a logical volume manager, allowing thin provisioning of volumes (called *filesets* in DCE/DFS) since space can be added to a storage region. Both of these systems can also maintain read-only copy-on-write clones as images of their active file systems.

DEC's AdvFS [10], available with the Tru64 operating system [5], provides a storage pooling concept atop which separate file systems (filesets) are allocated. It allows not only for media growth, and thus complete thin provisioning, but also media shrinkage as well. Disks can be added to or deleted from an AdvFS storage pool. It, too, provides for read-only copy-on-write images of filesets, also called clones.

SunTM ZFS is the file system with the closest match

to FlexVol functionality. ZFS provides multiple directory trees (file systems) in a storage pool and supports both read-only snapshots and read-write clones [26]. ZFS does not use a two-level block addressing scheme in the manner of FlexVol volumes. Instead, all file systems, snapshots, and clones in a ZFS storage pool use storage pool address, the equivalent of aggregate-level PVBNs [25]. Thus, ZFS avoids some of the overheads in WAFL, such as the need to store dual VBN mappings and to perform block allocation in both the container file and the aggregate. As we have shown, these overheads are quite modest in WAFL, and the resulting indirection facilitates the introduction of new functionality.

IBM's Storage Tank is an example of another class of file system. It stores file data and file system metadata on separate, shared storage devices [19]. Treating the data as a whole, Storage Tank implements multiple file system images ("containers") that can grow or shrink, and active and snapshot file systems can share pointers to storage blocks. Read-write clones are not provided.

The basic FlexVol idea of virtualizing file systems by creating a file system inside a file can be implemented using standard tools on commodity operating systems. Both the BSD and Linux® operating systems support the creation of a block device backed by a file. By formatting these devices as subsidiary file systems, an administrator can achieve a result similar to FlexVol volumes. This mode of operation would support thin provisioning by using a sparse backing file. It would not, however, provide many of the other optimizations and enhancements available with FlexVols—dual VBNs, storage deallocation by hole punching, free behind, clone volumes, etc.

There are many block-oriented storage systems that provide virtualization functionality similar to FlexVol volumes. Logical volume managers (LVMs) such as Veritas™ Volume Manager [27] and LVM2 in Linux [17] allow the dynamic allocation of logical disk volumes from pools of physical disk resources. Some volume managers also include support for read-only snapshots and read-write clones. Many mid-range and high-end disk arrays [4, 8] provide similar features, essentially implementing volume management internally to the device and exporting the resulting logical disk volumes to hosts via block protocols such as FCP or iSCSI.

Since volume managers and disk arrays provide block-oriented storage virtualization, their implementation differs substantially from file-based virtualization such as FlexVol volumes. Volume managers do not support allocation at the fine grain of a file system (4KB in WAFL), so copy-on-write allocation for snapshots and clones is handled in larger units—often several megabytes. Similarly, file-system-level knowledge allows WAFL to determine when blocks are no longer in use (e.g., because they belonged to a deleted file) so it

can transfer free space from a FlexVol volume back to its aggregate, enabling features such as thin provisioning.

7 Conclusion

FlexVol volumes separate the management of physical storage devices from the management of logical data sets, reducing the management burden of a file server and allowing administrators to match data set size to user and application needs. The virtualization of file volumes also provides increased flexibility for many routine management tasks, such as creating or resizing volumes or dynamically dividing physical storage among volumes. The FlexVol architecture enables many new features, including FlexClone volumes and thin provisioning of volumes. Using a few simple optimizations, we provided this expanded functionality at low cost. Many operations see no overhead when using FlexVol volumes. Even a workload with a broad functional profile, such as SFS, only shows a modest performance degradation of 4%—much less than the performance gain achieved with each new generation of hardware.

8 Acknowledgments

We thank Sudheer Miryala, the lead development manager on the FlexVol project, and Blake Lewis, who was the Technical Director for WAFL and provided many significant insights in numerous discussions. We also wish to thank the many others at NetApp—far too many to list here—whose inspiration and hard work made FlexVol volumes a reality. Finally, we thank the anonymous reviewers for their thoughtful comments.

References

- [1] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of SIGMETRICS 2007: Measurement and Modeling of Computer Systems*, pages 289–300, San Diego, CA, June 2007.
- [2] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 223–238, San Jose, CA, February 2008.
- [3] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, December 1994.

- [4] Gustavo A. Castets, Daniel Leplaideur, J. Alcino Bras, and Jason Galang. *IBM Enterprise Storage Server, SG24-5465-01*. IBM Corporation, October 2001.
- [5] Matthew Check, Scott Fafrak, Steven Hancock, Martin Moore, and Gregory Yates. *Tru64 UNIX System Administrators Guide*. Digital Press, 2001.
- [6] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode File System. In *Proceedings of the Winter 1992 USENIX Conference*, pages 43–60, San Francisco, CA, January 1992.
- [7] Peter Corbett, Bob English, Atul Goel, Tomislav Granac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, San Francisco, CA, March 2004.
- [8] EMC Corporation. EMC Symmetrix DMX Architecture Product Description Guide. http://www.emc.com/products/systems/interstitial/inter_c1011.jsp, March 2004.
- [9] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, Redmond, WA, 1994.
- [10] Steven Hancock. *Tru64 UNIX File System Administration Handbook*. Digital Press, 2000.
- [11] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Francisco, CA, January 1994.
- [12] John Howard. An Overview of the Andrew File System. Technical Report CMU-ITC-88-062, Information Technology Center, Carnegie-Mellon University, 1988.
- [13] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Michael West. Scale and Performance in a Distributed System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [14] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are Disks the Dominant Contributor for Storage Failures? In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2008.
- [15] Michael Kazar. Synchronization and Caching Issues in the Andrew File System. Technical Report CMU-ITC-88-063, Information Technology Center, Carnegie-Mellon University, 1988.
- [16] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Ben A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Antony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum File System Architectural Overview. In *Proceedings of the Summer 1990 USENIX Conference*, pages 151–164, Anaheim, CA, June 1990.
- [17] A.J. Lewis. LVM HOWTO. <http://www.linux.org/docs/ldp/howto/LVM-HOWTO>, 2006.
- [18] Marshall K. McKusick, William N. Joy, Samuel J. Lefler, and Robert S. Fabry. A Fast File System for UNIX. *Computer Systems*, 2(3):181–197, 1984.
- [19] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank—A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.
- [20] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File-System Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, pages 117–129, Monterey, CA, January 2002.
- [21] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.
- [22] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 71–84, San Diego, CA, June 2000.
- [23] Robert Sidebotham. VOLUMES: the Andrew File System data structuring primitive. In *Proceedings of the European UNIX Systems User Group*, pages 473–480, September 1986.
- [24] SPEC SFS (System File Server) Benchmark. <http://www.spec.org/osg/sfs97r1>, 1997.
- [25] Sun Microsystems, Inc. ZFS On-Disk Specification. <http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf>, 2006.
- [26] Sun Microsystems, Inc. Solaris ZFS Administration Guide. <http://www.opensolaris.org/os/community/zfs/docs/zfsadmin.pdf>, 2008.
- [27] Symantec Corporation. Veritas Volume Manager Administrator's Guide. http://sfdoccentral.symantec.com/sf/5.0/solaris/pdf/vxvm_admin.pdf, 2006.
- [28] Mark Wittle and Bruce E. Keith. LADDIS: The Next Generation in NFS File Server Benchmarking. In *Proceedings of the Summer 1993 USENIX Technical Conference*, pages 111–128, Cincinnati, OH, June 1993.

NetApp, Data ONTAP, FlexClone, FlexVol, RAID-DP, SnapMirror, Snapshot, and WAFL are trademarks of NetApp, Inc. in the United States and/or other countries. Linux is a registered trademark of Linus Torvalds. Intel is a registered trademark of Intel Corporation. Solaris and Sun are trademarks of Sun Microsystems, Inc. Veritas is a trademark of Symantec Corporation or its affiliates in the U.S. and other countries. UNIX is a registered trademark of The Open Group. All other brands or products are trademarks or registered trademarks of their respective holders and should be treated as such.

Fast, Inexpensive Content-Addressed Storage in Foundation

Sean Rhea*,
Meraki, Inc.

Russ Cox, Alex Pesterev*
MIT CSAIL

Abstract

Foundation is a preservation system for users' personal, digital artifacts. Foundation preserves all of a user's data and its dependencies—fonts, programs, plugins, kernel, and configuration state—by archiving nightly snapshots of the user's entire hard disk. Users can browse through these images to view old data or recover accidentally deleted files. To access data that a user's current environment can no longer interpret, Foundation boots the disk image in which that data resides under an emulator, allowing the user to view and modify the data with the same programs with which the user originally accessed it.

This paper describes Foundation's archival storage layer, which uses content-addressed storage (CAS) to retain nightly snapshots of users' disks indefinitely. Current state-of-the-art CAS systems, such as Venti [34], require multiple high-speed disks or other expensive hardware to achieve high performance. Foundation's archival storage layer, in contrast, matches the storage efficiency of Venti using only a single USB hard drive. Foundation archives disk snapshots at an average throughput of 21 MB/s and restores them at an average of 14 MB/s, more than an order of magnitude improvement over Venti running on the same hardware. Unlike Venti, Foundation does not rely on the assumption that SHA-1 is collision-free.

1 Introduction

We are “living in the midst of digital Dark Ages” [23]. As computer users increasingly store their most personal data—photographs, diaries, letters—only in digital form, they practically ensure that it will be unavailable to future generations [28].

Considering only the cost of storage, this state of affairs seems inexcusable. A half-terabyte USB hard drive now costs just over \$100, while reliable remote storage has become an inexpensive commodity: Amazon's S3 service [1], for example, charges only \$0.15/GB/month.

Alas, mere access to the bits of old files does not imply the ability to interpret those bits. Some file formats may be eternal—JPEG, perhaps—but most are ephemeral. Furthermore, the interpretation of a particular file may require a non-trivial set of support files. Consider, for example, the files needed to view a web page in its original form: the HTML itself, the fonts it uses, the right web browser and plugins. The browser and plugins themselves depend on a

particular operating system, itself depending on a particular hardware configuration. In the worst case, a user in the distant future might need to replicate an entire hardware-software stack to view an old file as it once existed.

Foundation is a system that preserves users' personal digital artifacts regardless of the applications with which they create those artifacts and without requiring any preservation-specific effort on the users' part. To do so, it permanently archives nightly snapshots of a user's entire hard disk. These snapshots contain the complete software stack needed to view a file in bootable form: given an emulator for the hardware on which that stack once ran, a future user can view a file exactly as it was. To limit the hardware that future emulators must support, Foundation confines users' environments to a virtual machine. Today's virtual machine monitor thus serves as the template for tomorrow's emulator.

Using emulation for preservation is not a new idea (see, e.g. [15, 35, 38]), but by archiving a complete image of a user's disk, Foundation captures *all* of the user's data, applications, and configuration state as a single, *consistent* unit. By archiving a new snapshot every night, Foundation prevents the installation of new applications from interfering with a user's ability to view older data—e.g., by overwriting the shared libraries on which old applications depend with new and incompatible versions [8]. Users view each artifact using the most recent snapshot that correctly interprets that artifact. There is no need for them to manually create an emulation environment particular to each artifact, or even to choose in advance which artifacts will be preserved.

Of course, such comprehensive archiving is not without risk: the cost of storing nightly snapshots of users' disks indefinitely may turn out to be prohibitive. On the other hand, the Plan 9 system archived nightly snapshots of its file system on a WORM jukebox for years [32, 33], and the subsequent Venti system [34] drastically reduced the storage required for those archives by using content-addressed storage (CAS) [18, 44] to automatically identify and coalesce duplicate blocks between snapshots.

The Plan 9 experience, and our own experience using a 15-disk Venti system to back up the main file server of a research group at MIT, convinced us that content-addressed storage was a promising technique for reducing Foundation's storage costs. Venti, however, requires multiple, high-performance disks to achieve acceptable archival throughput, an unacceptable cost in the consumer setting in which we intend to deploy Foundation. A new design seemed necessary.

*Work done while at Intel Research, Berkeley

The core contribution of this paper is the design, implementation, and evaluation of Foundation's content-addressed storage system. This system is inspired by Venti [34], but we have modified the Venti design for consumer use, replacing Venti's expensive RAID array and high speed disks with a single, inexpensive USB hard drive. Foundation achieves high archival throughput on modest hardware by using a Bloom filter to quickly detect new data and by making assumptions about the structure of duplicate data—assumptions we have verified using over a year of Venti traces. Our evaluation of the resulting system shows that Foundation achieves read and write speeds an order of magnitude higher than Venti on the same hardware.

While we built Foundation for digital preservation, content-addressed storage is useful in other contexts, and we believe Foundation will enable other applications of CAS that were previously confined to the enterprise to enter the consumer space. As an anecdotal example, we note that within our own households, most computers share a large percentage of their files—digital photos, music files, mail messages, etc. A designer of a networked household backup server could easily reduce its storage needs by adopting Foundation as its storage system.

In this paper, however, we focus on the CAS layer itself. To ground the discussion, Section 2 provides background on the Foundation system as a whole. Sections 3–5 then present the main contributions of the paper—the design, implementation, and evaluation of Foundation's content-addressed storage layer. Section 6 surveys related work, Section 7 describes future work, and Section 8 concludes.

2 Background: Foundation

Figure 1 shows the major components of a Foundation system. The host operating system runs on the raw hardware, providing a local file system and running Foundation. Users work inside the active VM, which runs a conventional OS like Windows XP or Linux atop Foundation's *virtual machine monitor* (VMM). The VMM stores virtual machine state (disk contents and other metadata) in the local file system. Every night, Foundation's *virtual machine archiver* takes a real-time snapshot of the active VM's state, storing the snapshot in the *CAS layer*.

In addition to taking nightly snapshots of the VM's state, the VM archiver also provides read-only access to previously-archived disk images. The VMM uses this functionality to boot past images; the figure shows an archived VM snapshot running in a separate VM. As a convenience, Foundation provides a *file system snapshot server* that interprets archived disk images, presenting each day's file system snapshot in a synthetic file tree (like Plan 9's dump file system [32] or NetApp's .snapshot directories [17]) that VMs can access over

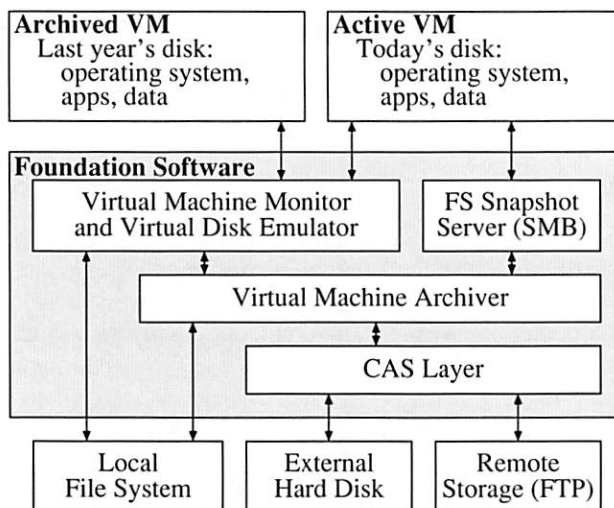


Figure 1: Foundation system components. A Foundation user works inside the active VM, which is archived daily to an external hard disk and (optionally) a remote location. Foundation presents archival file system data using SMB and enables users to interpret obsolete file formats by booting VM snapshots from days or years past.

SMB.¹ A user finds files from May 1, 1999, for example, in `/snapshot/1999/05/01/`. This gives the active VM access to old data, but it cannot guarantee that today's system will be able to understand the data. The fallback of being able to boot the VM image provides that guarantee.

Foundation's CAS layer provides efficient storage of nightly snapshots taken by the VM archiver. The CAS layer stores archived data on an inexpensive, external hard disk. Users can also configure the CAS layer to replicate its archives onto a remote FTP server for fault tolerance. To protect users' privacy, the CAS layer encrypts data before writing to the external hard drive or replicating it. It also signs the data and audits the local disk and replica to detect corruption or tampering.

As a simple optimization, Foundation interprets the partition table and file systems on the guest OS's disk to identify any swap files or partitions. It treats such swap space as being filled with zeros during archival.

The remainder of this section discusses the components of Foundation in detail, starting with the VMM and continuing through the VM archiver and CAS layer.

2.1 Virtual Machine Monitor

Foundation uses VMware Workstation as its virtual machine monitor. Foundation configures VMware to store the contents of each emulated disk as a single, contiguous file, which we call the disk image. VMware's snapshot

¹Providing the snapshot tree requires that Foundation interpret the partition table and file systems on the guest OS's disk. Foundation interprets ext2/3 and NTFS using third-party libraries. Support for other file systems is easy to add, and if no such library exists, a user can always boot the VM image to access a file.

facility stores the complete state of a VM at a particular instant in time. Foundation uses this facility to acquire consistent images of the VM's disk image.

To take a snapshot, VMware reopens the disk image read-only and diverts all subsequent disk writes to a new partial disk image. To take a second snapshot, VMware reopens the first partial disk image read-only and diverts all subsequent disk writes to a second partial disk image. A sequence of snapshots thus results in a stack of partial disk images, with the original disk image at the bottom. To read a sector from the virtual disk, VMware works down the stack (from the most recent to the oldest partial disk image, ending with the original disk) until it finds a value for that sector [2].

To discard a snapshot, VMware removes the snapshot's partial disk image from the stack and applies the writes contained in that image to the image below it on the stack. Notice that this procedure works for discarding any snapshot, not just the most recent one.

The usual use of snapshots in VMware is to record a working state of the system before performing a dangerous operation. Before installing a new application, for example, a user can snapshot the VM, rolling back to the snapshotted state if the installation fails.

2.2 Virtual Machine Archiver

Foundation uses VMware's snapshot facility both to obtain consistent images of the disk and to track daily changes between such images.

Foundation archives consistent images of the disk as follows. First, the VM archiver directs VMware to take a snapshot of the active VM, causing future disk writes to be diverted into a new partial disk image. The archiver then reads the now-quietest original disk image, storing it in the CAS layer along with the VM configuration state and metadata about when the snapshot was taken. Finally, the virtual machine archiver directs VMware to discard the snapshot. Using a snapshots in this way allows Foundation to archive a consistent disk image without suspending the VM or interrupting the user.

Note that the above algorithm requires Foundation to scan the entire disk image during the nightly archival process. For a large disk image, this process can take considerable time. For this reason, Foundation makes further use of the VMM's snapshotting facility to track daily changes in the disk image as illustrated in Figure 2.

Between snapshots, the VM archiver keeps VMware in a state where the bottom disk image on the stack corresponds to the last archived snapshot (say, snapshot k), with VMware recording writes since that snapshot in a partial disk image. To take and archive snapshot $k+1$, the VM archiver takes another VMware snapshot, causing VMware to push a new partial disk image onto the stack. The VM archiver then archives only those blocks written

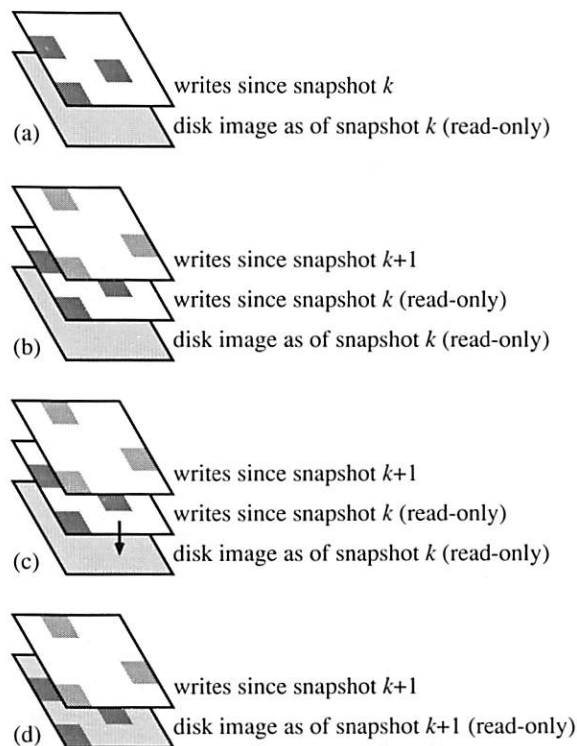


Figure 2: The VMware disk layers when the VM archiver archives disk image snapshot $k+1$. (a) Before the snapshot. The base VMware disk corresponds to snapshot k , already archived; since then VMware has been saving disk writes in a partial disk image layered on top of the base image. (b) During the snapshot archival process. The VM archiver directed VMware to create a new snapshot, $k+1$, adding a second partial disk image to the disk stack. The earlier partial disk image contains only the disk sectors that were written between snapshots k and $k+1$. The VM archiver saves these using the CAS layer. (c) After the snapshot has been archived. The VM archiver directs VMware to discard snapshot k . VMware applies the writes from the corresponding partial disk image to the base disk image and (d) discards the partial disk image.

to the now read-only partial disk image for snapshot k . Once those blocks have been saved, the VM archiver directs VMware to discard snapshot k , merging those writes into the base disk image.

Using VM snapshots in this way allows Foundation to archive a consistent image of the disk without blocking the user during the archival process. However, because Foundation does not yet use VMware's "SYNC driver" to force the file system into a consistent state before taking a snapshot, the guest OS may need to run a repair process such as *fsck* when the user later boots the image. An alternate approach would archive the machine state and memory as well as the disk, and "resume", rather than boot, old snapshots. We have not yet explored the additional storage costs of this approach.

2.3 CAS Layer

Foundation's CAS layer provides the archival storage service that the VM archiver uses to save VM snapshots. This service provides a simple *read/write* interface: passing a disk block to *write* returns a short handle, and *read*, when passed the handle, returns the original block. Internally, the CAS layer coalesces duplicate writes, so that writing the same block multiple times returns the same handle and only stores one copy of the block. Coalescing duplicate writes makes storing many snapshots feasible; the additional storage cost for a new snapshot is proportional only to its new data. The rest of this paper describes the CAS layer in detail.

3 CAS Layer Design

Foundation's CAS layer is modeled on the Venti [34] content-addressed storage server, but we have adapted the Venti algorithms for use in a single-disk system and also optionally eliminated the assumption that SHA-1 is free of collisions, producing two operating modes for Foundation: *compare-by-hash* and *compare-by-value*.

In this section, we first review Venti and then introduce Foundation's two modes. We also discuss the expected disk operations used by each algorithm, since those concerns drove the design.

3.1 Venti Review

The Venti content-addressed storage server provides SHA-1-addressed block storage. When a client writes a disk block, Venti replies with the SHA-1 hash of the block's contents, called a *score*, that can be used to identify the block in future read requests. The storage server provides read/write access to disk blocks, typically ranging in size from 512 bytes up to 32 kilobytes. Venti clients conventionally store larger data streams in hash trees (also known as Merkle trees [29]).

As illustrated in Figure 3, Venti stores blocks in an append-only data log and maintains an index that maps blocks' scores to their offsets in the log. Venti implements this index as a on-disk hash table, where each bucket contains (score, log offset) pairs for a subsection of the 160-bit score space. Venti also maintains two write-through caches in memory: the *block cache* maps blocks' scores to the blocks' values, and the *index cache* maps blocks' scores to the blocks' log offsets.

Figure 4(a) gives pseudocode for the Venti read and write operations. To satisfy a read of a block with a given score, Venti first looks in the block cache. If the block is not found in the block cache, Venti looks up the block's offset in the log, first checking the index cache and then the index itself. If Venti finds a log offset for the block, it reads the block from the log and returns the block. Otherwise, it returns an error (not shown). Writes are handled similarly. Venti first checks to see if it has an existing

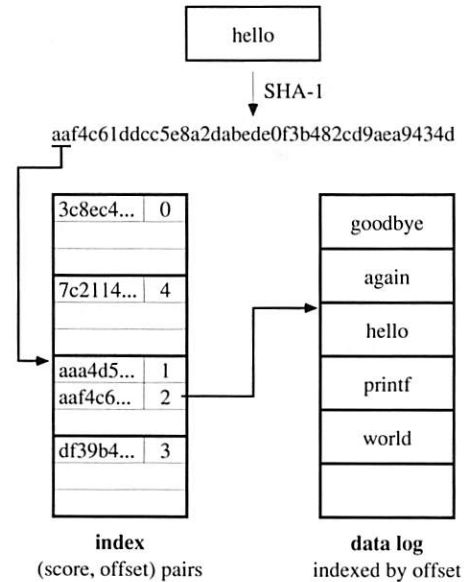


Figure 3: Venti's on-disk data structures. The SHA-1 hash of a data block produces a *score*, the top bits of which are used as a bucket number in the index. The index bucket contains an index entry—a (score, offset) pair—indicating the offset of the corresponding block in the append-only data log.

offset for the block using the two in-memory caches and then the index, returning immediately if so. Otherwise, it appends the block to the log and updates its index and caches before returning.

Note that Venti must read at least one block of its index to satisfy a read or write that misses in both the block and index caches. Because blocks' scores are essentially random, each such operation necessitates at least one seek to read the index. In a single-disk system, these seeks limit throughput to *block size/seek time*. The Venti prototype striped its index across eight dedicated, high-speed disks so that it could run eight times as many seeks at once.

3.2 Foundation: Compare-by-Hash Mode

While Venti was designed to provide archival service to many computers, Foundation is aimed at individual consumers and cannot afford multiple disks to mask seek latency. Instead, Foundation stores both its archive and index on a single, inexpensive USB hard drive and uses additional caches to improve archival throughput.²

In compare-by-hash mode, Foundation optimizes for two request types: sequential reads (reading blocks in the order in which they were originally written) and fresh writes (writing new blocks).

Foundation stores its log as a collection of 16 MB *arenas* and stores for each arena a separate *summary* file that lists all of the (score, offset) pairs the arena contains.³ To

²An alternative approach—storing the index in Flash memory—would eliminate seek cost for reads but greatly increase it for writes. Current Flash memories require around 40 ms for random writes.

³This design was inspired by Venti's log arenas. We do not know

(a) Venti

```
// Return block named by score.
read(score):
    if(data = blockcache.get(score))
        return data;
    offset = lookupscore(score);
    data = log.read(offset);
    blockcache.put(score, data);
    return data;

// Write data, returning score.
write(data):
    score = SHA1(data);
    if(lookupscore(score))
        return score;
    offset = log.write(data);
    index.write(score, offset);
    indexcache.put(score, offset);
    blockcache.put(score, data);
    return score;

// Return log offset for score.
lookupscore(score):
    if(offset = indexcache.get(score))
        return offset;
    if(offset = index.read(score))
        indexcache.put(score, offset);
        return offset;
    return nil;
```

(b) Foundation: Compare by Hash

```
// Return block named by score.
read(score):
    if(data = blockcache.get(score))
        return data;
    offset = lookupscore(score);
    data = log.read(offset);
    blockcache.put(score, data);
    return data;

// Write data, returning score.
write(data):
    score = SHA1(data);
    if(lookupscore(score))
        return score;
    offset = log.write(data);
    indexbuffer.write(score, offset);
    indexcache.put(score, offset);
    blockcache.put(score, data);
    bloomfilter.put(score);
    return score;

// Return log offset for score.
lookupscore(score):
    if(!bloomfilter.get(score))
        return nil;
    if(offset = indexcache.get(score))
        return offset;
    if(offset = index.read(score))
        sum = log.summary(offset);
        indexcache.put(sum);
        return offset;
    return nil;
```

(c) Foundation: Compare by Value

```
// Read block named by offset.
read(offset):
    if(data = blockcache.get(offset))
        return data;
    // No lookup score!
    data = log.read(offset);
    blockcache.put(offset, data);
    return data;

// Write data, returning offset.
write(data):
    score = hash(data);
    if(offset = lookupdata(data, score))
        return offset;
    offset = log.write(data);
    indexbuffer.write(score, offset);
    indexcache.put(score, offset);
    blockcache.put(score, data);
    bloomfilter.put(score);
    return offset;

// Return log offset for data.
lookupdata(data, score):
    if(!bloomfilter.get(score))
        return nil;
    for(offset in indexcache.get(score))
        if(read(offset) == data)
            return offset;
    for(offset in index.read(score))
        if(offset in indexcache.get(score))
            continue;
        if(read(offset) == data)
            sum = log.summary(offset);
            indexcache.put(sum);
            return offset;
    return nil;
```

Figure 4: Algorithms for reading and writing blocks in (a) Venti and Foundation’s (b) compare-by-hash and (c) compare-by-value modes. Italics in (b) mark differences from (a): the addition of a Bloom filter, the use of a buffer to batch index updates in *write*, and the loading of entire arena summaries into the index cache after a miss in *lookupscore*. Italics in (c) mark differences from (b): the use of log offsets to identify blocks, the use of an insecure hash function to identify potential duplicate writes, the possibility of multiple index entries for a given score, and the need to check existing blocks’ contents against new data in *lookupdata*.

take advantage of the spatial locality inherent in sequential reads, each time Foundation reads its on-disk index to find the log offset of some block, it loads and caches the entire summary for the arena that spans the discovered offset. Reading this summary costs an additional seek. This cost pays off in subsequent reads to the same arena, as Foundation finds the log offsets of the affected blocks in the cached summary, avoiding seeks in the on-disk index.

Figure 5 summarizes the costs in disk operations of each path through the pseudocode in Figure 4. In addition to sequential reads and fresh writes, the figure shows

whether Venti’s design was inspired by the log segments of LFS [36].

costs for out-of-order reads (reading blocks in a different order than that in which they were written), sequential duplicate writes (writing already-written blocks in the same order in which they were originally written), and out-of-order duplicate writes (writing already-written blocks in a different order).

Note that for out-of-order disk reads and for the first disk read in each arena, compare-by-hash mode is slower than Venti, as it performs an additional seek to read the arena summary. In return, Foundation performs subsequent reads at the full throughput of the disk. Section 5 shows that this tradeoff improves overall throughput in real workloads.

(a) Venti	(b) Foundation: by Hash	(c) Foundation: by Value
Out-of-order read		
seek+read index bucket seek+read log block	seek+read index bucket seek+read arena summary seek+read log block	seek+read log block
<i>Cost</i> 2 seeks + $I+L$ reads	3 seeks + $I+L+S$ reads	1 seek + L reads
Sequential read		
same as out-of-order	if(first block in arena) seek+read index bucket seek+read arena summary seek to log block read log block	if(first block in arena) seek to log block read log block
<i>Cost</i> 2 seeks + $I+L$ reads	$(1/A) \times (3 \text{ seeks} + I+S \text{ reads}) + L \text{ reads}$	$(1/A) \times 1 \text{ seek} + L \text{ reads}$
Out-of-order duplicate write		
seek+read index bucket	seek+read index bucket seek+read arena summary	seek+read index bucket $(C+1) \times \text{seek+read log block}$ seek+read arena summary
<i>Cost</i> 1 seek + I reads	(2 seeks + $I+S$ reads)	$(C+3 \text{ seeks} + I+(C+1)L+S \text{ reads})$
Out-of-order duplicate write — index entry cached		
no disk operations	no disk operations	seek + read log block
<i>Cost</i> none	none	1 seek + L reads
Sequential duplicate write		
same as out-of-order	if(first block in arena) same as out-of-order	if(first block in arena) same as out-of-order else read log block
<i>Cost</i> 1 seek + I reads	$(1/A) \times (2 \text{ seeks} + I+S \text{ reads})$	$(1/A) \times (C+3 \text{ seeks} + I+(C+1)L+S \text{ reads})$ + $(1-1/A) \times L \text{ reads}$
Fresh write		
seek+read index bucket seek+write log block seek+write index bucket	if(Bloom filter false positive) seek+read index bucket seek to end of log write log block if(index buffer full) flush index buffer	if(Bloom filter false positive) seek+read index bucket $C \times \text{seek+read log block}$ seek to end of log write log block if(index buffer full) flush index buffer
<i>Cost</i> 3 seeks + I reads + $L+I$ writes	$B \times (2 \text{ seeks} + I \text{ reads}) + L \text{ writes}$ + $(1/W) \times 1 \text{ index buffer flush}$	$B \times (C+2 \text{ seeks} + I+CL \text{ reads}) + L \text{ writes}$ + $(1/W) \times 1 \text{ index buffer flush}$

Figure 5: Disk operations required to handle the five different read/write cases. A is the number of blocks per arena, B is the probability of a Bloom filter false positive, C is the probability of a hash collision, I is the size of an index bucket, L is the size of a log data block, S is the size of an arena summary, and W is the size of the write buffer in index entries.

On fresh writes, Venti performs three seeks: one to read the index and determine the write is fresh, one to append the new block to the log, and one to update the index with the block's log offset (see Figure 5).

Foundation eliminates the first of these three seeks by maintaining an in-memory Bloom filter [6] summarizing the all of the scores in the index. A Bloom filter is a randomized data structure for testing set membership. Us-

ing far less memory than the index itself, the Bloom filter can check whether a given score is in the index, answering either “probably yes” or “definitely no”. A “probably yes” answer for a score that is *not* in the index is called a *false positive*. Using enough memory, the probability of a false positive can be driven arbitrarily low. (Section 4 discusses sizing of the Bloom filter.) By first checking the in-memory Bloom filter, Foundation determines that a write

is fresh without reading the on-disk index in all but a small fraction of these writes.

By buffering index updates, Foundation also eliminates the seek Venti performs to update the index during a fresh write. When this buffer fills, Foundation applies the updates in a single, sequential pass over the index. Fresh writes thus proceed in two phases: one phase writes new data to the log and fills the index update buffer; a second phase flushes the buffer. During the first phase, Foundation performs no seeks within the index; all disk writes sequentially append to the end of the log. In return, it must occasionally pause to flush the index update buffer; Section 5 shows that this tradeoff improves overall write throughput in real workloads.

3.3 Foundation: Compare-by-Value Mode

In compare-by-value mode, Foundation does not assume that SHA-1 is collision-free. Instead, it names blocks by their log offsets, and it uses the on-disk index only to identify *potentially* duplicate blocks, comparing each pair of potential duplicates byte-by-byte.

While we originally investigated this mode due to (in our opinion, unfounded) concerns about cryptographic hash collisions (see [5, 16] for a lively debate), we were surprised to find that its overall write performance was close to that of compare-by-hash mode, despite the added comparisons. Moreover, compare-by-value is *always* faster for reads, as naming blocks by their log offsets completely eliminates index lookups during reads.

The additional cost of compare-by-value mode can be seen in the *lookupdata* function in Figure 4(c). For each potential match Foundation finds in the index cache or the index itself, it must read the corresponding block from the log and perform a byte-by-byte comparison.

For sequential duplicate writes, Foundation reads the blocks for these comparisons sequentially from the log. Although these reads consume disk bandwidth, they require a seek only at the start of each new arena. For out-of-order duplicate writes, however, the relative cost of compare-by-value is quite high. As shown in Figure 5, Venti and compare-by-hash mode complete out-of-order duplicate writes without any disk activity at all, whereas compare-by-value mode requires a seek per write.

On the other hand, hash collisions in compare-by-value mode are only a performance problem (as they cause additional reads and byte-by-byte comparisons), not a correctness one. As such, compare-by-value mode can use smaller, faster (and less secure) hash functions than Venti and compare-by-hash. Our prototype, for example, uses the top four bytes of an MD4 hash to select an index block, and stores the next four bytes in the block itself. Using four bytes is enough to make collisions within an index block rare (see Section 4). It also increases the number of entries that fit in the index write buffer, making flushes

less frequent, and decreases the index size, making flushes faster when they do occur. Both changes improve the performance of fresh writes.

Section 5 presents a detailed performance comparison between Venti and Foundation's two modes.

3.4 Compare-by-Hash vs. Compare-by-Value

It is worth asking what other disadvantages, other than decreased write throughput, compare-by-value incurs in naming blocks by their log offsets.

The Venti paper lists five benefits of naming blocks by their SHA-1 hashes: (1) blocks are immutable: a block cannot change its value without also changing its name; (2) writes are idempotent: duplicate writes are coalesced; (3) the hash function defines a universal name space for block identifiers; (4) clients can check the integrity of data returned by the server by recomputing the hash; and (5) the immutability of blocks eliminates cache coherence problems in a replicated or distributed storage system.

Benefits (1), (2), and (3) apply also to naming blocks by their log offsets, as long as the log is append-only. Log writes are applied at the client in Foundation—the remote storage service is merely a secondary replica—so (5) is not an issue. Foundation's compare-by-value mode partially addresses benefit (4) by cryptographically signing the log, but naming blocks by their hashes, as in compare-by-hash mode, still provides a more end-to-end guarantee.

Our own experience with Venti also provides one obscure, but interesting case in which naming blocks by their SHA-1 hashes provides a small but tangible benefit. A simultaneous failure of both the backup disk and the remote replica may result in the loss of some portion of the log, after which reads for the lost blocks will fail. In archiving the user's current virtual machine, however, Foundation may encounter many of the lost blocks. When it does so, it will append them to the log as though they were new, but because it names them by their SHA-1 hashes, they will have the same names they had before the failure. As such, subsequent reads for the blocks will begin succeeding again. In essence, archiving current data can sometimes "heal" an injured older archive. We have used this technique successfully in the past to recover from corrupted Venti archives.

4 Implementation

The Foundation prototype consists of just over 14,000 lines of C++ code. It uses VMware's VIX library [3] to take and delete VM snapshots. It uses GNU parted, libext2, and libntfs to read interpret disk images for export in the /snapshot tree.

The CAS layer stores its arenas, arena summaries, and index on an external USB hard disk. To protect against loss of or damage to this disk, the CAS layer can be configured to replicate the log arenas over FTP using libcurl.

Providers such as dot5hosting.com currently lease remote storage for as little as \$5/month for 300 GB of space. While this storage may not be as reliable as that offered by more expensive providers, we suspect that fault-tolerance obtained through the combination of one local and one remote replica is sufficient for most users' needs. The CAS layer does not replicate the arena summaries or index, as it can recreate these by scanning the log.

While users may trust such inexpensive storage providers as a secondary replica for their data, they are less likely to be comfortable entrusting such providers with the contents of their most private data. Moreover, the external hard drive on which Foundation stores its data might be stolen. The CAS layer thus encrypts its log arenas to protect users' privacy, and it cryptographically signs the arenas to detect tampering.

For good random-access performance, our implementation uses a hierarchical HMAC signature and AES encryption in counter mode [11] to sign and encrypt arenas. The combination allows Foundation to read, decrypt, and verify each block individually (i.e., without reading, decrypting, and verifying the entire arena in which a block resides). Foundation implements its hierarchical HMAC and counter-mode AES cipher using the OpenSSL project's implementations of AES and HMAC. (It also uses OpenSSL's SHA-1 and MD4 implementations to compute block hashes.)

Foundation uses the file system in user-space (FUSE) library to export its `/snapshot` tree interface to the host OS. The guest OS then mounts the host's tree using the SMB protocol. To provide the archived disk images for booting under VMware, Foundation uses a loopback NFS server to create the appearance of a complete VMware virtual machine directory, including a `.vmx` file, the read-only disk image, and a `.vmdk` file that points to the read-only image as the base disk while redirecting new writes to an initially empty snapshot file.

By default, the prototype uses a 192 MB index cache—with 128 MB reserved for buffering index writes and the remaining 64 MB managed in LRU order—and a 1 MB block cache. It also caches 10 arena summaries in LRU order, using approximately 10 MB more memory. The prototype stores index entries with 6 bytes for the log offset, 20 bytes for the score in compare-by-hash mode, and 4 bytes for the score in compare-by-value mode. It sizes the index to average 90% full for a user-configurable expected maximum log size. In compare-by-hash mode, a 100 GB log yields a 5.6 GB index. The same log yields a 2.2 GB index in compare-by-value mode. The prototype relocates index block overflow entries using linear probing. It sizes its Bloom filter such that half its bits will be set when the log is full and lookups see a 0.1% false positive rate. For a 100 GB log, the Bloom filter consumes 361 MB of memory. To save memory, the prototype loads

the Bloom filter only during the nightly archival process; it is not used during read-only operations such as booting an image or mounting the `/snapshot` tree.

Currently, the Foundation prototype uses 512-byte log blocks to maximize alignment between data stored from different file systems. Using a 512-byte block size also aligns blocks with file systems within a disk, as the master boot record (MBR), for example, is only 512-bytes long, and the first file system usually follows the MBR directly. That said, per-block overheads are a significant factor in Foundation's performance, so we are considering increasing the default block size to 4 kB (now the default for most file systems) and handling the MBR as a special case.

5 Evaluation

To evaluate Foundation, we focus on the performance of saving and restoring VM snapshots, which corresponds directly to the performance of the CAS layer.

The most important performance metric for Foundation is how long it takes to save the VM disk image each night. Many users suspend or power down their machines at night; a nightly archival process that makes them wait excessively long before doing so is a barrier to adoption. (We envision that snapshots are taken automatically as part of the shutdown/sleep sequence.) We are also concerned with how long it takes to boot old system images and recover old file versions from the `/snapshot` tree, though we expect such operations to be less frequent than nightly backups, so their performance is less critical.

We evaluate Foundation's VM archiver in two experiments. First, we analyze the performance of the CAS layer on microbenchmarks in three ways: using the disk operation counts from Figure 5, using a simulator we wrote, and using Foundation itself. These results give insight into Foundation's performance and validate the simulator's predictions. Second, we measure Foundation's archival throughput under simulation on sixteen months of nightly snapshots using traces derived from our research group's own backups.

In both experiments, we compare Foundation in compare-by-hash and compare-by-value mode with a third mode that implements the algorithms described in the Venti paper. Making the comparison this way rather than using the original Venti software allows us to compare the algorithms directly, without worrying about other variables, such as file system caches, that would be different between Foundation and the actual Venti. (Although we do not present the results here, we have also implemented Foundation's compare-by-hash improvements in Venti itself and obtained similar speedups.)

5.1 Experimental setup

We ran our experiments on a Lenovo Thinkpad T60 laptop with a 2 GHz Intel Core 2 Duo Processor and 2 GB of

	— Expected Throughput (kB/s) —			— Actual Throughput (kB/s) —		
	Venti	Foundation By-Hash	By-Value	Venti	Foundation By-Hash	By-Value
out-of-order read	18	7.4	37	15	4.8	19
sequential read	18	29,000	33,000	76	13,000	16,000
out-of-order duplicate write	36	9.2	7.4	79	6.0	5.2
index entry cached	39,000	39,000	37	22,000	22,000	19
sequential duplicate write	36	39,000	29,000	78	23,000	16,000
fresh write	12	4,000	8,100	37	3,800	7,100
without write buffer flush	n/a	11,000	11,000		7,900	8,400

Figure 6: Predicted and actual sustained performance, in MB/s, of the three systems on the cases listed in Figure 5 using the hardware described in Section 5.1. The actual performance of our Venti implementation is faster than predicted, because operating system readahead eliminates some seeks. The actual performance of Foundation is slightly slower than predicted because of unmodeled per-block overheads: using a 4096-byte block size (instead of 512 bytes) matches predictions more closely.

RAM. The laptop runs Ubuntu 7.04 with a Linux 2.6.20 SMP kernel. The internal hard disk is a Hitachi Travelstar 5K160 with an advertised 11 ms seek time and 64 MB/s sustained read/write throughput, while the external disk is a 320 GB Maxtor OneTouch III with an advertised 9 ms seek time and 33 MB/s sustained read/write throughput.

Since Foundation uses both disks through the host OS’s file system, we measured their read and write throughput through that interface using the Unix `dd` command. For read throughput, we copied a 2.2 GB file to `/dev/null`; for write throughput, we copied 2.2 GB of `/dev/zero` into a half-full partition. The Hitachi sustained 38.5 MB/s read and 32.2 MB/s write throughput; the Maxtor sustained 32.2 MB/s read and 26.5 MB/s write throughput.

To measure average seek time plus rotational latency through the file system interface, we wrote a small C program that seeks to a random location within the block device using the `lseek` system call and reads a single byte using the `read` system call. In 1,000 such “seeks” per drive, we measured an average latency of 15.0 ms on the Hitachi and 13.6 ms on the Maxtor. The system was otherwise idle during both our throughput and seek tests.

The simulator uses the disk speeds we measured and the same parameters (cache sizes, etc.) as our implementation. Rather than store the Bloom filter directly, it assumes a 0.1% probability of a false positive.

5.2 Microbenchmarks

To understand Foundation’s performance, we consider the disk operations required for each of the six read or write cases shown in Figure 5. For each case, we count the number of seeks and the amount of data read from and written to the disk. From these and the disk parameters measured and reported above, we compute the speed of each algorithm in each case. Figure 6 shows the predicted performance and the performance of the prototype. (The simulated performance matches the predictions made using the equations in Figure 5 exactly.)

In both prediction and in reality, compare-by-hash is significantly faster than Venti for sequential accesses, at

the cost of slowing out-of-order accesses, which load arena summaries that end up not being useful. Compare-by-value reads faster than compare-by-hash, since it avoids the index completely, but it handles duplicate writes slower, since it must compare each potential duplicate to previously-written data from the log.

The most dramatic difference between compare-by-hash and compare-by-value is the case of an out-of-order duplicate write for which the index entry cache has a corresponding record, but the block cache does not. In this case, Venti and compare-by-hash can declare the write a duplicate without any disk accesses, while compare-by-value must load the data from disk, resulting in dramatically lower throughput. (The throughput for Venti and compare-by-hash is limited only by the bandwidth of the local disk in this case.)

Sequential duplicate writes are fast in both Foundation modes. In compare-by-hash mode, Foundation is limited by the throughput of the local disk containing the snapshot. The arena summaries needed from the external disk are only 5% the size of the snapshot itself. In compare-by-value mode, Foundation must read the snapshot from the local disk and compare it again previously-written data from the log disk. Having two disks arms here is the key to good performance: on a single-disk system the performance would be hurt by seeks between the two streams.

Fresh writes proceed at the same speed in both Foundation modes except for the index buffer flushes. Because index entries are smaller in compare-by-value mode, the 128 MB buffer holds more entries and needs to be flushed less frequently: after every 4 GB of fresh writes rather than every 2.3 GB. At that rate, index flushes are still an important component of the run time. Using a larger buffer size or a larger data block size would reduce the flush frequency, making the two modes perform more similarly.

The predictions match Foundation’s actual performance to within a factor of 2.25, and the relative orderings are all the same. Foundation is slower than predicted because the model does not account for time spent encrypting, signing, verifying, and decrypting the log; time spent

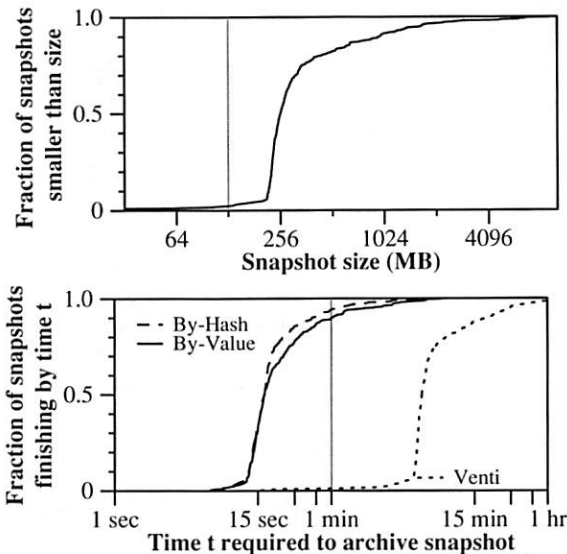


Figure 7: Distribution of sizes and write times for 400 nightly snapshots of one of our research group’s home directory disks.

compressing and decompressing blocks; and constant (per 512-byte block) overheads in the run-time system. Using 4096-byte blocks and disabling encryption, compression, and authentication yields performance that matches the predictions more accurately.

5.3 Trace-driven Simulation

We do not yet have long-term data from using Foundation, but as mentioned earlier, our research group takes nightly physical backups of its central file server using a 15-disk Venti server. The backup program archives entire file system images, using the file system block size as the archival block size. We extracted over a year of block traces from each of the file server’s 10 disks. These traces contain, for each night, a list of the disk blocks changed from the previous night, along with the blocks’ SHA-1 hashes. We annotated each trace with the data log offsets each block would have been stored at if data from the disk were the only data in a Venti or Foundation server. We then ran the traces in our simulator to compare the two Foundation operating modes and the Venti mode.

To conserve space, we discuss the results from only one of the traces here. The relative performance of the three algorithms, however, is consistent across traces. The disk for the trace we chose hosts the home directories of four users. The trace covers 400 days. When the trace starts, the disk has 41.7 GB of data on it; when the trace ends, the disk has 69.9 GB of data on it. The parameters for the simulation are the same as described in Section 5.1, except that blocks are 32 kB, to match the traces, rather than 512 bytes as in Foundation.

The most important metric is the duration of the nightly snapshot process. Figure 7 plots the distributions of snapshot sizes and completion times. Even though 95% of

snapshots are larger than 128 MB, the vast majority of snapshots—90% for compare-by-value and 94% for compare-by-hash—finish in a minute or less.

Figure 8 breaks down the average performance of a snapshot backup. The differences in snapshot speed—849 kB/s for Venti, 20,581 kB/s for compare-by-hash, and 15,723 kB/s for compare-by-value—are accounted for almost entirely by the time spent seeking in the external disk. Foundation’s use of the Bloom filter and arena summaries reduces the number of seeks required in compare-by-hash mode by a factor of 240 versus Venti. Compare-by-value mode reintroduces some seeks by reading log blocks to decide that writes are duplicates during lookup.

To access archived data, Foundation users will either use the file system snapshot server or boot an archived VM. In both cases, the relevant parts of disk can be read as needed by the file system browser or the VMM.

In many cases, Foundation can satisfy such reads quickly. Comparing the measured performance of Foundation in Figure 6 with the performance of our test system’s internal hard drive, we note that Foundation’s compare-by-value mode is only 1.8 times slower for out-of-order reads and 2.2 times slower for sequential reads. However, in eliminating duplicate data during writes, Foundation may introduce additional seeks into future reads, since the blocks of a disk image being read may originally have been stored as part of other disk images earlier in the log. We call this problem *fragmentation*.

Unfortunately, we do not have traces of the reads requests serviced by our research group’s Venti server, so it is difficult to simulate to what degree users will be affected by such fragmentation in practice. As an admittedly incomplete benchmark, however, we simulate reading entire disk images for each snapshot. We return to the fragmentation problem in Section 7.2.

Figure 9 summarizes the performance of reading full disk images. Again the differences in performance are almost entirely due to disk seeks: 739 minutes seeking for Venti, 41 minutes seeking for compare-by-hash, and 35 minutes seeking for compare-by-value. Since compare-by-value eliminates index lookups during read, its seeks are all within the data log. Such seeks are due to fragmentation, and for compare-by-value mode they account for the entire difference between the predicted performance of sequential reads in Figure 6 with the simulated performance in Figure 9.

In compare-by-value mode, since the block identifiers are log offsets, the reads could be reordered to reduce the amount of seeking. As a hypothetical, the column labeled “Sorted” shows the performance if the block requests were first sorted in increasing log offset. This would cut the total seek time from 35 minutes to 8 minutes, also improving the number of block cache hits by a factor of 24. Although making a list of every block may not be realis-

	Venti	Foundation	
		By-Hash	By-Value
average snapshot write speed	849 kB/s	20,581 kB/s	15,723 kB/s
average snapshot time	648.4 s	26.7 s	35.0 s
... reading external disk	4.1 s	2.2 s	4.5 s
... writing external disk	21.6 s	20.0 s	19.8 s
... seeking in external disk	622.3 s	2.6 s	10.7 s
... waiting for local disk; external disk idle	0.4 s	2.0 s	0.0 s
average # of external disk seeks	46,099	192	792
... to index data	31,415	133	133
... to log data	14,684	58	658
average # of lookup calls; these do ...	17,196	2,526	2,526
... # of index seeks (also # of index reads)	16,731	73	73
... # of log seeks	0	0	442
... # of log reads	0	0	2,482

Figure 8: Statistics gathered while writing 400 nightly snapshots, in simulation. The average snapshot size is 537 MB. Because the VMM identifies which blocks have changed since the previous snapshot, on average only 78.5 MB of blocks are duplicates. There are, however, occasional large spikes of duplicates. 9 of the 400 nights contain over 1 GB of duplicate blocks; 2 contain over 5 GB.

	Venti	By-Hash	Foundation		
			Unsorted	Sort-1024	Sorted
average disk image restore speed	1,271 kB/s	13,894 kB/s	15,309 kB/s	20,842 kB/s	28,940 kB/s
average disk image restore time	775 min	71 min	64 min	47 min	34 min
... reading external disk	36 min	30 min	30 min	29 min	26 min
... seeking in external disk	739 min	41 min	35 min	18 min	8 min
average # of block cache hits	9,660	9,660	9,660	31,203	232,597
average # of index entry cache hits	222,936	1,824,023	0	0	0
average # of external disk seeks	3,283,167	182,337	153,853	79,495	35,218
... to index data	1,613,802	25,431	0	0	0
... to log data	1,669,365	156,906	153,853	79,495	35,218
average # of external disk reads	3,450,540	1,849,454	1,836,738	1,815,196	1,613,802
... of index data	1,613,802	12,716	0	0	0
... of log data	1,836,738	1,836,738	1,836,738	1,815,196	1,613,802

Figure 9: Statistics gathered while reading disk images of 400 nightly snapshots, in simulation. The average disk image is 56 GB.

tic, a simple heuristic can realize much of the benefit. The column labeled “Sort-1024” shows the performance when 1024 reads at a time are batched and sorted before being read from the log. This simple optimization cuts the seek time to 18 minutes, while still improving the number of block cache hits by a factor of 3.2.

6 Related Work

Related Work in Preservation Most preservation work falls into one of two groups. (The following description is simplified somewhat; see Lee et al. [24] for a detailed discussion.) The first group (e.g. [12, 14, 37, 41]) proposes archiving a limited set of popular file formats such as JPEG, PDF, or PowerPoint. This restriction limits the digital artifacts that can be preserved to those that can be encoded in a supported format. In contrast, Foundation preserves both the applications and configuration state needed to view both popular and obscure file formats.

In the case that a supported format becomes obsolete, this first group advocates automated “format migration”,

in which files in older formats are automatically converted to more current ones. Producing such conversion routines can be difficult: witness PowerPoint’s inability to maintain formatting between its Windows and Mac OS versions. Furthermore, perfect conversion is sometimes impossible, as between image formats that use lossy compression. Rather than migrate formats forward in time, Foundation enables travel back in time to the environments in which old formats can be interpreted.

The second group of preservationists (e.g. [15, 38]) advocates emulating old hardware and/or operating systems in order to run the original applications with which users viewed older file formats. Foundation uses emulation, but recognizes that simply preserving old applications and operating systems is not enough. Often, the rendering of a digital artifact is dependent on configuration state, optional shared libraries, or particular fonts. A default Firefox installation, for example, may not properly display a web page that contains embedded video, non-standard fonts, or Flash animations. Foundation captures all such

state by archiving full disk images, but it limits the hardware that must be emulated to boot such images by confining users' daily environments within a VM.

An offshoot of the emulation camp proposes the construction of emulators specifically for archival purposes. Lorie proposed [27] storing with each digital artifact a program for interpreting the artifact; he further proposed that such programs be written in the language of a Universal Virtual Computer (UVC) that can be concisely specified and for which future emulators are easy to construct. Ford has proposed [13] a similar approach, but using an x86 virtual machine with limited OS support as the emulation platform. Foundation differs from these two systems in that it archives files with the same OS kernel and programs originally used to view them, rather than require the creation of new ones specific to archival purposes.

Internet Suspend/Resume (ISR) [22] and Machine Bank [43] use a VM to suspend a user's environment on one machine and resume it on another. SecondSite [10] and Remus [9] allow resumption of services at a site that suffers a power failure by migrating the failed site's VMs to a remote site. Like these systems, Foundation requires that a user's environment be completely contained within a VM, but for a different purpose: it allows the "resumption" of state from arbitrarily far in the past.

Related Work in Storage Hutchinson et al. [19] demonstrated that physical backup can sustain higher throughput than logical backup, but noted several problems with physical backup. First, since bits are not interpreted as they are backed up, the backup is not portable; Foundation provides portability by booting the entire image in an emulator. Second, it is hard to restore only a subset of a physical backup; Foundation interprets file system structures to provide the /snapshot tree, allowing users to recover individual files using standard file system tools. Third, obtaining a consistent image is difficult; Foundation implements copy-on-write within the VMM to do so, but other tools, such as the Linux's Logical Volume Manager (LVM) [25] could be used instead. Finally, incremental backups are hard; addressing blocks by their hashes as in Venti solves this problem.

The SUNDR secure network file system [26] also uses a Venti-like content-addressed storage server but uses a different solution than Foundation to reduce index seeks. SUNDR saves all writes in a temporary disk buffer without deciding whether they are duplicate or fresh and then batches both the index searches to determine freshness and the index updates for the new data. Foundation avoids the temporary data buffer by using the Bloom filter to determine freshness quickly, buffering only the index entries for new writes, and never the content.

Microsoft Single-Instance Store (SIS) [7] identifies and collates files with identical contents within a file system, but rather than coalescing duplicates on creation, SIS in-

stead finds them using a background "groveler" process.

A number of past file systems have provided support for sharing blocks between successive file versions using copy-on-write (COW) [17,31,39,42]. These systems capture duplicate blocks between versions of the same file, but they fail to identify and coalesce duplicate blocks that enter the file system through different paths—as when a user downloads a file twice, for example. Moreover, they cannot coalesce duplicate data from multiple, distinct file systems; a shared archival storage server built on such systems would not be as space-efficient as one built on CAS.

Peabody [20] implements time travel at the disk level, making it possible to travel back in time to any instant and get a consistent image. Chronus [45] used Peabody to boot old VMs to find a past configuration error. Peabody uses a large in-memory content-addressed buffer cache [21] to coalesce duplicate writes. Because it only looks in the buffer cache, it cannot guarantee that all duplicate writes are coalesced. In contrast, Foundation is careful to find all duplicate writes.

LBFS [30] chooses block boundaries according to blocks' contents, rather than using a fixed block size, in order to better capture changes that shift the alignment of data within a file. Foundation is agnostic as to how block boundaries are chosen and could easily be adapted to do the same.

Time Machine [40] uses incremental logical backup to store multiple versions of a file system. It creates the first backup by logically mirroring the entire file system tree onto a remote drive. For each subsequent backup, Time Machine creates another complete tree on the remote drive, but it uses hard links to avoid re-copying unchanged files. Unlike Foundation, then, Time Machine cannot efficiently represent single-block differences. Even if a file changes in only one block, Time Machine creates a complete new copy on the remote drive. The storage cost of this difference is particularly acute for applications such as Microsoft Entourage, which stores a user's complete email database as a single file.

7 Future Work

The Foundation CAS layer is already a fully functioning system; it has been in use as one author's only backup strategy for six months now. In using it on a daily basis, however, we have discovered two interesting areas for future work: storage reclamation and fragmentation.

7.1 Storage Reclamation

Both the Plan 9 experience and our own experience with Venti seem to confirm our hypothesis that, in practice, content-addressed storage is sufficiently space-efficient that users can retain nightly disk snapshots indefinitely.

Nonetheless, it is not difficult to imagine usage patterns that would quickly exhaust the system's storage. Consider,

for example, a user that rips a number of DVDs onto a laptop to watch during a long business trip, but shortly afterwards deletes them. Because the ripped DVDs were on the laptop for several nights, Foundation is likely to have archived them, and they will remain in the user's archive. After a number of such trips, the archive disk will fill.

One solution to this problem would allow users to selectively delete snapshots. This solution is somewhat risky, in that a careless user might delete the only snapshot that is able to interpret a valued artifact. We suspect that users would be even more frustrated, however, by having to add disks to a system that was unable to reclaim space they felt certain was being wasted.

Like Venti, Foundation encodes the metadata describing which blocks make up a snapshot as a Merkle tree and stores interior nodes of this tree in the CAS layer. To simplify finding a particular snapshot within the log, Foundation also implements a simple *system catalog* as follows. After writing a snapshot, Foundation writes the root of the snapshot's Merkle tree along with the time at which it took the snapshot to a file that it then archives in the CAS layer. It repeats this process after writing each subsequent snapshot, appending the new snapshot's root and time to the existing list and re-archiving the list. The last block in Foundation's log is thus always the root of the latest version of the system catalog.

Conceptually, deleting a snapshot resembles garbage collection in programming languages or log cleaning in LFS. First, the CAS layer writes a new version of the system catalog that no longer points to the snapshot. Then, the system reclaims the space used by blocks that are no longer reachable from any other catalog entry. A more recent snapshot, for example, may still point to some block in the deleted snapshot.⁴

Interestingly, the structure of Foundation's log makes identifying unreferenced blocks particularly efficient: as a natural result of the log being append-only, all pointers within the log point "backwards". Garbage collection can thus proceed in a single, sequential pass through the log using an algorithm developed by Armstrong and Virding for garbage collecting immutable data structures in the Erlang programming language [4].

The algorithm works as follows. Starting at the most recent log entry and scanning backwards, it maintains a list of "live" blocks initialized from the pointers in the system catalog. Each time it encounters a live block, it deletes that block from its list. If the block is a metadata block that contains pointers to other blocks, it adds these pointers to its list. If the algorithm encounters a block that is not in its list, then there are no live pointers to that block later in the log, and since all pointers point backwards, the algorithm can reclaim the block's space immediately. The

⁴Here Foundation differs from LFS, which collects all blocks not pointed to by the most recent version.

system can also use this algorithm incrementally: starting from the end of the log, it can scan backward until "enough" space has been reclaimed, and then stop.

The expensive part of the Erlang algorithm is maintaining the list of live blocks. If references to many blocks occur much later in the log than the blocks themselves, this list could grow too large to fit in memory. We note, however, that a conservative version of the collector could use a Bloom filter to store the list of live blocks. Although false positives in the filter would prevent the algorithm from reclaiming some legitimate garbage, its memory usage would be fixed at the size of the Bloom filter.

Finally, to reclaim the space used by an unreferenced block, Foundation can simply rewrite the log arena in which the block occurs without the block, using an atomic rename to replace the old arena. Because this rewriting shifts the locations of other blocks in the arena, an extra pass is required in compare-by-value mode, where blocks' names are their locations in the log: the system must scan from the rewritten arena to the tail of the log, rewriting pointers to the affected arena as it goes. In compare-by-hash mode, however, blocks' names are independent of their locations in the log, so no extra pass is required.

7.2 Fragmentation

Most of our current work on the Foundation CAS layer has focused on reducing the number of seeks within the index. Having done so, however, we have noticed a potential secondary bottleneck: seeks within the data log itself. Consider the case of an archived snapshot made up of one block from each of all of the arenas in the log. Even if no seeks were required to determine the location of the snapshot's blocks, reading the snapshot would still incur one seek (into the appropriate arena) per block.

We have come to call this problem *fragmentation*. We have not yet studied the sources of fragmentation in detail. In our experience so far it is a visible problem, but not a serious one. We simply see some slowdown in reading later versions of disk images as they evolve over time.

Unfortunately, unlike the seeks within the system's index, seeks due to fragmentation cannot be eliminated; they are a fundamental consequence of coalescing duplicate writes (the source of Foundation's storage efficiency). We suspect that it also exists in file systems that perform copy-on-write snapshots, such as WAFL [17], although we have not found any reference to it in the literature.

We do note that fragmentation can be eliminated in any one snapshot, at the expense of others, by copying all of the blocks of that snapshot into a contiguous region of the log. If the system also removes the blocks from their original locations, this process resembles the "defragmentation" performed by a copying garbage collector. We are thus considering implementing within Foundation a version of the Erlang algorithm discussed above that reclaims

space by copying live data, rather than deleting dead data, in order to defragment more recently archived (and presumably, more frequently accessed) snapshots.

One other potential motivation for defragmenting more recent snapshots in this manner is that it will likely improve the write throughput of compare-by-value mode, since the blocks it compares against while writing are unlikely to change their ordering much between snapshots.

8 Conclusion

Foundation's approach to preservation—archiving consistent, nightly snapshots of a user's entire hard disk—is a straight-forward, application-independent approach to automatically capturing all of a user's digital artifacts and their associated software dependencies. Archiving these snapshots using content-addressed storage keeps the system's storage cost proportional to the amount of new data users create and eliminates duplicates that file-system-based techniques, such as copy-on-write, would miss. Using the techniques described in this paper, CAS achieves high throughput on remarkably modest hardware—a single USB hard disk—improving on the read and write throughput achieved by an existing, state-of-the-art CAS system on the same hardware by an order of magnitude.

9 Acknowledgments

This work benefits from useful discussions with Eric Brewer, Greg Ganger, David Gay, William Josephson, Michael Kaminsky, Jinyang Li, and Petros Maniatis.

References

- [1] Amazon simple storage service (S3). <http://www.amazon.com/gp/browse.html?node=16427261>, 2007.
- [2] VMware virtual machine disk format (VMDK) specification. <http://www.vmware.com/interfaces/vmdk.html>, 2007.
- [3] VMware VIX API. <http://www.vmware.com/support/developer/vix-api/>, 2007.
- [4] J. Armstrong and R. Virding. One pass real-time generational mark-sweep garbage collection. In *Intl. Workshop on Memory Management*, 1995.
- [5] J. Black. Compare-by-hash: A reasoned analysis. In *USENIX Annual Tech. Conf.*, 2006.
- [6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [7] W. Bolosky, S. Corbin, D. Goebel, and J. Douceur. Single instance storage in Windows 2000. In *4th USENIX Windows Symp.*, 2000.
- [8] R. Chen. Getting out of DLL Hell. *Microsoft TechNet*, Jan. 2007.
- [9] B. Cully et al. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.
- [10] B. Cully and A. Warfield. SecondSite: disaster protection for the common server. In *HotDep*, 2006.
- [11] M. Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. *NIST Special Publication 800-38A*, Dec. 2001.
- [12] P. Festa. A life in bits and bytes (interview with Gordon Bell). <http://news.com.com/2008-1082-979144.html>, Jan. 2003.
- [13] B. Ford. VXA: A virtual architecture for durable compressed archives. In *FAST*, 2005.
- [14] J. Gemmell, G. Bell, and R. Lueder. MyLifeBits: a personal database for everything. *CACM*, 49(1):88–95, Jan. 2006.
- [15] S. Granger. Emulation as a digital preservation strategy. *D-Lib Magazine*, 6(10), Oct. 2000.
- [16] V. Henson. An analysis of compare-by-hash. In *HotOS*, 2003.
- [17] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Conf.*, 1994.
- [18] J. Hollingsworth and E. Miller. Using content-derived names for configuration management. In *ACM SIGSOFT Symposium on Software Reusability*, 1997.
- [19] N. Hutchinson et al. Logical vs. physical file system backup. In *OSDI*, 1999.
- [20] C. B. M. III and D. Grunwald. Peabody: The time travelling disk. In *MSST*, 2003.
- [21] C. B. M. III and D. Grunwald. Content based block caching. In *MSST*, 2006.
- [22] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *WMCSA*, 2002.
- [23] T. Kuny. A digital dark ages? Challenges in the preservation of electronic information. In *63rd IFLA General Conference*, 1997.
- [24] K.-H. Lee, O. Slattery, R. Lu, X. Tang, and V. McCrary. The state of the art and practice in digital preservation. *Journal of Research of the NIST*, 107(1):93–106, Jan–Feb 2002.
- [25] A. Lewis. LVM HOWTO. <http://tldp.org/HOWTO/LVM-HOWTO/>, 2006.
- [26] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.
- [27] R. Lorie. A methodology and system for preserving digital data. In *ACM/IEEE Joint Conf. on Digital Libraries*, 2002.
- [28] C. Marshall, F. McCown, and M. Nelson. Evaluating personal archiving strategies for Internet-based information. In *IS&T Archiving*, 2006.
- [29] R. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1988.
- [30] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, Oct. 2001.
- [31] OpenSolaris. What is ZFS? <http://opensolaris.org/os/community/zfs/whatis/>, 2007.
- [32] R. Pike et al. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.
- [33] S. Quinlan. A cached WORM file system. *Software—Practice and Experience*, 21(12):1289–1299, 1991.
- [34] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *FAST*, 2002.
- [35] T. Reichherzer and G. Brown. Quantifying software requirements for supporting archived office documents using emulation. In *ICDL*, 2006.
- [36] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.
- [37] D. Rosenthal, T. Lipkis, T. Robertson, and S. Morabito. Transparent format migration of preserved web content. *D-Lib Magazine*, 11(1), Jan. 2005.
- [38] J. Rothenberg. Ensuring the longevity of digital documents. *Scientific American*, 272(1):42–47, Jan. 1995.
- [39] D. Santry et al. Deciding when to forget in the Elephant file system. In *SOSP*, 1999.
- [40] J. Siracusa. Mac OS X 10.5 Leopard: the Ars Technica review. <http://arstechnica.com/reviews/os/mac-os-x-10-5-ars/14>, 2007.
- [41] M. Smith. Eternal bits. *IEEE Spectrum*, 42(7):22–27, July 2005.
- [42] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in versioning file systems. In *FAST*, 2003.
- [43] S. Tang, Y. Chen, and Z. Zhang. Machine Bank: Own your virtual personal computer. In *IPDPS*, 2007.
- [44] A. van Hoff, J. Giannandrea, M. Hapner, S. Carter, and M. Medin. The HTTP distribution and replication protocol. Technical Report NOTE-drp-19970825, W3C, 1997.
- [45] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.

Adaptive File Transfers for Diverse Environments

Himabindu Pucha*, Michael Kaminsky[‡], David G. Andersen*, and Michael A. Kozuch[‡]

*Carnegie Mellon University and [‡]Intel Research Pittsburgh

Abstract

This paper presents *dsync*, a file transfer system that can dynamically adapt to a wide variety of environments. While many transfer systems work well in their specialized context, their performance comes at the cost of generality, and they perform poorly when used elsewhere. In contrast, *dsync* adapts to its environment by intelligently determining which of its available resources is the best to use at any given time. The resources *dsync* can draw from include the sender, the local disk, and network peers. While combining these resources may appear easy, in practice it is difficult because these resources may have widely different performance or contend with each other. In particular, the paper presents a novel mechanism that enables *dsync* to aggressively search the receiver's local disk for useful data without interfering with concurrent network transfers. Our evaluation on several workloads in various network environments shows that *dsync* outperforms existing systems by a factor of 1.4 to 5 in one-to-one and one-to-many transfers.

1 Introduction

File transfer is a nearly universal concern among computer users. Home users download software updates and upload backup images (or delta images), researchers often distribute files or file trees to a number of machines (e.g. conducting experiments on PlanetLab), and enterprise users often distribute software packages to cluster or client machines. Consequently, a number of techniques have been proposed to address file transfer, including simple direct mechanisms such as FTP, “swarming” peer-to-peer systems such as BitTorrent [4], and tools such as *rsync* [22] that attempt to transfer only the small delta needed to re-create a file at the receiver.

Unfortunately, these systems fail to deliver optimal performance due to two related problems. First, the solutions typically focus on one particular resource strategy to the exclusion of others. For example, *rsync*'s delta approach will accelerate transfers in low-bandwidth environments when a previous version of the file exists in the current directory, but not when a useful version exists in a sibling directory or, e.g., `/tmp`. Second, existing solutions typically do not adapt to unexpected environments. As an example, *rsync*, by default, always inspects previous

file versions to “accelerate” the transfer—even on fast networks when such inspections contend with the write portion of the transfer and *degrade* overall performance.

This paper presents *dsync*, a file(tree) transfer tool that overcomes these drawbacks. To address the first problem, *dsync* opportunistically uses all available sources of data: the sender, network peers, and similar data on the receiver's local disk. In particular, *dsync* includes a framework for locating relevant data on the local disk that might accelerate the transfer. This framework includes a pre-computed index of blocks on the disk and is augmented by a set of heuristics for *extensively* searching the local disk when the cache is out-of-date. *dsync* addresses the second problem, the failure of existing file transfer tools to accommodate diverse environments, by constantly monitoring resource usage and adapting when necessary. For example, *dsync* includes mechanisms to throttle the aggressive disk search if either the disk or CPU becomes a bottleneck in accepting data from the network.

Those two principles, opportunistic resource usage and adaptation, enable *dsync* to avoid the limitations of previous approaches. For example, several peer-to-peer systems [6, 1, 4, 11, 19] can efficiently “swarm” data to many peers, but do not take advantage of the local filesystem(s). The Low Bandwidth File System [13] can use *all* similar content on disk, but must maintain an index and can only transfer data from one source. When used in batch mode, *rsync*'s diff file can be sent over a peer-to-peer system [11], but in this case, all hosts to be synchronized must have identical initial states.

dsync manages three main resources: the network, the disk, and CPU. The network (which we assume can provide all of the data) is *dsync*'s primary data source, but *dsync* can choose to spend CPU and disk bandwidth to locate relevant data on the local filesystem. However, *dsync* also needs these CPU resources to process incoming packets and this disk bandwidth to write the file to permanent storage. Therefore, *dsync* adaptively determines at each step of the transfer which of the receiver's local resources can be used without introducing undue contention by monitoring queue *back-pressure*. For example, *dsync* uses queue information from its disk writer and network reader processes to infer disk availability (Section 4). When searching the receiver's disk is viable,

dsync must continuously evaluate whether to identify additional candidate files (by performing directory `stat` operations) or to inspect already identified files (by reading and hashing the file contents). *dsync* prioritizes available disk operations using a novel cost/benefit framework which employs an extensible set of heuristics based on file metadata (Section 5.2). As disk operations are scheduled, *dsync* also submits data chunk transfer requests to the network using the expected arrival time of each chunk (Section 5.3).

Section 6 presents the implementation of *dsync*, and Section 7 provides an extensive evaluation, demonstrating that *dsync* performs well in a wide range of operating environments. *dsync* achieves performance near that of existing tools on the workloads for which they were designed, while drastically out-performing them in scenarios beyond their design parameters. When the network is extremely fast, *dsync* correctly skips any local disk reads, performing the entire transfer from the network. When the network is slow, *dsync* discovers and uses relevant local files quickly, outperforming naive schemes such as breadth-first search.

2 Goals and Assumptions

The primary goal of *dsync* is to correctly and efficiently transfer files of the order of megabytes under a wide range of operating conditions, maintaining high performance under dynamic conditions and freeing users from manually optimizing such tasks. In our current use-model, applications invoke *dsync* on the receiver by providing a description of the file or files to be fetched (typically end-to-end hashes of files or file trees). *dsync* improves transfer performance (a) by exploiting content on the receiver's local disk that is similar to the files currently being transferred, and (b) by leveraging additional network peers that are currently downloading or are sources of the files being transferred (*identical sources*) or files similar to the files being transferred (*similar sources*).

Typical applications that benefit from similar content on disk include updating application and operating system software [5], distributed filesystems [13], and backups of personal data. In addition to the above workloads, network similarity is applicable in workloads such as email [7], Web pages [8, 12], and media files [19]. Peer-to-peer networks, mirrors, and other receivers engaged in the same task can also provide additional data sources.

The following challenges suggest that achieving this goal is not easy, and motivate our resulting design.

Challenge 1: Correctly use resources with widely varying performance. Receivers can obtain data from many resources, and *dsync* should intelligently schedule these resources to minimize the total transfer time. *dsync* should work well on networks from a few kilobits/sec to gigabits/sec and on disks with different I/O loads (and

thus different effective transfer speeds).

Challenge 2: Continuously adapt to changes in resource performance. Statically deciding upon the relative merits of network and disk operations leads to a fragile solution. *dsync* may be used in environments where the available network bandwidth, number of peers, or disk load changes dramatically over the course of a transfer.

Challenge 3: Support receivers with different initial filesystem state. Tools for synchronizing multiple machines should not require that those machines start in the same state. Machines may have been unavailable during prior synchronization attempts, may have never before been synchronized, or may have experienced local modification subsequent to synchronization. Regardless of the cause, a robust synchronization tool should be able to deal with such differences while remaining both effective and efficient.

Challenge 4: Do not require resources to be set up in advance. For example, if a system provides a pre-computed index that tells *dsync* where it can find desired data, *dsync* will use it (in fact, *dsync* builds such an index as it operates). However, *dsync* does not *require* that such an index exists or is up-to-date to operate correctly and effectively.

Assumption 1: At least one correct source. Our system requires at least one sender to transfer data directly; the receiver can *eventually* transfer all of the data (correctly) from the sender.

Assumption 2: Short term collision-resistant hashes. *dsync* ensures correctness by verifying the cryptographic hash of all of the data it draws from any resource. Our design requires that these hashes be collision-resistant for the duration of the transfer, but does not require long-term strength.

Assumption 3: Unused resources are “free”. We assume that the only cost of using a resource is the lost opportunity cost of using it for another purpose. Thus, tactics such as using spare CPU cycles to analyze every file on an idle filesystem are “free.” This assumption is not valid in certain environments, e.g., if saving energy is more important than response time; if the costs of using the network vary with use; or if using a resource degrades it (e.g., limited write cycles on flash memory). While these environments are important, our assumption about free resource use holds in many other important environments. Similarly, *dsync* operates by greedily scheduling at each receiver; we leave the issue of finding a globally optimal solution in a cooperative environment for future work.

Assumption 4: Similarity exists between files. Much prior work [7, 13, 5, 8, 12, 19] has shown that similarity exists and can be used to reduce the amount of data transferred over the network or stored on disk. A major benefit of *rsync* and related systems is their ability to

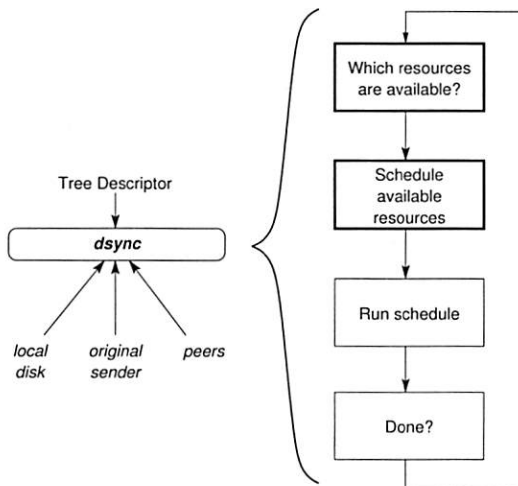


Figure 1: The *dsync* system

exploit this similarity.

3 Design Overview

dsync operates by first providing a description of the file or files to be transferred to the receiver. *dsync* receiver then uses this description to fetch the “recipe” for the file transfer. Like many other systems, *dsync* source divides each file or set of files (file tree) into roughly 16 KB chunks, and it hashes each chunk to compute a chunk ID.¹ Then, *dsync* creates a *tree descriptor* (TD), which describes the layout of files in the file tree, their associated metadata, and the chunks that belong to each file. The TD thus acts as a recipe that tells the receiver how to reconstruct the file tree.

Figure 1 shows a conceptual view of *dsync*’s operation. Given a TD (usually provided by the sender), *dsync* attempts to fetch the chunks described in it from several sources in parallel. In this paper, we focus on two sources: the network and the disk. As we show in Section 7, intelligent use of these resources can be critical to achieving high performance.

3.1 The Resources

dsync can retrieve chunks over the **network**, either directly from the original sender or from peers downloading the same or similar content. Our implementation of *dsync* downloads from peers using a BitTorrent-like peer-to-peer swarming system, SET [19], but *dsync* could easily use other similar systems [4, 1, 11]. Given a set of chunks to retrieve, SET fetches them using a variant of the rarest-chunk-first strategy [10].

dsync can also look for chunks on the receiver’s **local**

¹*dsync* uses Rabin fingerprinting [20] to determine the chunk boundaries so that most of the chunks that a particular file is divided into do not change with small insertions or deletions.

disk, similar to existing systems such as *rsync* [22]. First, *dsync* consults a pre-computed index if one is available. If the index is not available or if the chunk ID is not in the index (or the index is out-of-date), *dsync* tries to locate and hash files that are most likely to contain chunks that the receiver is trying to fetch. Note that the disk is used as an optimization: *dsync* validates all chunks found by indexing to ensure that the transfer remains correct even if the index is out-of-date or corrupt.

dsync requires **CPU** resources to compute the hash of data from the local disk and to perform scheduling computations that determine which chunks it should fetch from which resource.

3.2 Control Loop

Figure 1 depicts the main *dsync* control loop. *dsync* performs two basic tasks (shown with thicker lines in the figure) at every scheduling step in its loop: First, *dsync* determines which resources are available. Second, given the set of available resources, *dsync* schedules operations on those resources. An operation (potentially) returns a chunk. After assigning some number of operations to available resources, *dsync* executes the operations. This control loop repeats until *dsync* fetches all of the chunks, and is invoked every time an operation completes. Sections 4 and 5 provide more detail about the above tasks.

4 Avoiding Contention

dsync’s first task is to decide which resources are available for use: the network, the disk, or both. Deciding which resources to use (and which not to use) is critical because using a busy resource might contend with the ongoing transfer and might actually slow down the transfer.² For example, searching the local disk for useful chunks might delay *dsync* from writing chunks to disk if those chunks are received over a fast network. Spending CPU time to compute a more optimal schedule, or doing so too often, could delay reading data from the network or disk.

Back-pressure is the central mechanism that *dsync* uses to detect contention and decide which resources are available to use. *dsync* receives back-pressure information from the network and the disk through the explicit communication of queue sizes.³ While the disk exposes information to *dsync* indicating if it is busy, the network indicates how many requests it can handle without incurring unnecessary queuing. *dsync* also monitors its CPU consumption to determine how much time it can spend computing a schedule.

²Note that *dsync* tries to avoid resource contention within an ongoing transfer. Global resource management between *dsync* and other running processes currently remains the responsibility of the operating system, but is an interesting area for future work.

³Our experience agrees with that of others who have found that explicit queues between components makes it easier to behave gracefully under high load [23, 2].

Disk: *dsync* learns that the disk is busy from two queues:

- **Writer process queue:** *dsync* checks for disk back-pressure by examining the outstanding data queued to the writer process (a separate process that writes completed files to disk). If the writer process has pending writes, *dsync* concludes that the disk is overloaded and stops scheduling new disk read operations. (Writes are critical to the transfer process, so they are prioritized over read operations.)
- **Incoming network queue:** *dsync* also infers disk pressure by examining the incoming data queued in-kernel on the network socket. If there is a queue build-up, *dsync* stops scheduling disk read operations to avoid slowing the receiver.

Network: *dsync* tracks availability of network resources by monitoring the *outstanding data requests queue* on a per sender basis. If any sender's outstanding data queue size is less than the number of bytes the underlying network connection can handle when all the scheduled network operations are issued, *dsync* infers that the network is available.

CPU: *dsync* consumes CPU to hash the content of local files it is searching, and to perform the (potentially expensive) scheduling computations described in the following section. When searching the local filesystem, the scheduler must decide which of thousands of candidate files to search for which of thousands of chunks. Thus, *dsync* ensures that performing expensive re-computation of a new, optimized schedule using CPU resources does not slow down an on-going transfer using the following guidelines at every scheduling step.

- As it does for disk contention, *dsync* defers scheduling computations if there is build-up in the *incoming network queue*.
- To reduce the amount of computation, *dsync* only re-computes the schedule if more than 10% of the total number of chunks or number of available operations have changed. This optimization reduces CPU consumption while bounding the maximum estimation error (e.g., limiting the number of chunks that would be erroneously requested from the network when a perfectly up-to-date schedule would think that those chunks would be better served from the local disk).
- *dsync* limits the total time spent in recomputation to 10% of the current transfer time.
- Finally, the scheduler limits the total number of files that can be outstanding at any time. This limitation sacrifices some opportunities for optimization in favor of reducing the time spent in recomputation.

The experiments in Section 7 show the importance of avoiding resource contention when adapting to different environments. For example, on a fast network (gigabit LAN), failing to respond to back pressure means that disk and CPU contention impose up to a 4x slowdown vs. receiving only from the network.

5 Scheduling Resources

Once the control loop has decided which resources are available, *dsync*'s scheduler decides which operations to perform on those resources. The actual operations are performed by a separate plugin for each resource, as described in the next subsections. The scheduler itself takes the list of outstanding chunks and the results of any previously completed network and disk operations and determines which operations to issue in two steps. (A) First, the scheduler decides which disk operations to perform. (B) Second, given an ordering of disk operations, the scheduler decides which chunks to request from the network. At a high level, *dsync*'s scheduler opportunistically uses the disk to speed the transfer, knowing that it can ultimately fetch all chunks using the network.

This two-step operation is motivated by the different ways these resources behave. Storage provides an opportunity to accelerate network operations, but the utility of each disk operation is uncertain *a priori*. In contrast, the network can provide all needed chunks. A natural design, then, is to send to the disk plugin those operations with the highest estimated benefit, and to request from the network plugin those chunks that are least likely to be found on disk.

5.1 Resource Operations

Network operations are relatively straightforward; *dsync* instructs the network plugin to fetch a set of chunks (given the hashes) from the network. Depending on the configuration, the network plugin gets the chunks from the sender directly and/or from peers, if they exist.

Disk operations are more complex because *dsync* may not know the contents of files in the local file system. *dsync* can issue three operations to the disk plugin to find useful chunks on the disk.

CACHE operations access the chunk index (a collection of *<ChunkID, ChunkSize, filename, offset, file metadata>-tuples*), if one exists. **CACHE-CHECK** operations query to see if a chunk ID is in the index. If it is, *dsync* then `stat()`s the identified file to see if its metadata matches that stored in the index. If it does not, *dsync* invalidates all cache entries pointing to the specified file. Otherwise, *dsync* adds a potential **CACHE-READ** operation, which reads the chunk from the specified file and offset, verifies that it matches the given hash, and returns the chunk data.

HASH operations ask the disk plugin to read the specified file, divide it into chunks, hash each chunk to determine a chunk ID, compare each chunk ID to a list of needed chunk IDs, and return the data associated with any matches via a callback mechanism. Found chunks are used directly, and *dsync* avoids fetching them from the network. The identifiers and metadata of all chunks are cached in the index.

STAT operations cause the disk plugin to read a given directory and `stat()` each entry therein. STAT does not locate chunks by itself. Instead, it generates additional potential disk operations for the scheduler, allowing *dsync* to traverse the filesystem. Each file found generates a possible HASH operation, and each directory generates another possible STAT operation.

dsync also increases the estimated priority of HASHing the enclosing file from a CACHE-CHECK operation as the number of CACHE-READs from that file increase. This optimization reduces the number of disk seeks when reading a file with many useful chunks (Section 7.4).

5.2 Step A: Ordering Disk Operations

The scheduler estimates, for each disk operation, a cost to benefit ratio called the *cost per useful chunk delivered* or *CPC*. The scheduler estimates the costs by dynamically monitoring the cost of operations such as disk seeks or hashing chunks. The scheduler estimates the benefit from these operations using an extensible set of heuristics. These heuristics consider factors such as the similarity of the source filename and the name of the file being hashed, their sizes, and their paths in the filesystem, with files on the receiver's filesystem being awarded a higher benefit if they are more similar to the source file. The scheduler then schedules operations greedily by ascending CPC.⁴

The *CPC* for an operation, *OP*, is defined as:

$$\begin{aligned} CPC_{OP} &= \frac{\text{total cost of the operation}}{\text{expected num. of useful chunks delivered}} \\ &= T_{OP} / E[N_{delivered}] \end{aligned}$$

T_{OP} , the *total* operation cost, includes all costs associated with delivering the chunk to *dsync* memory, including the “wasted” work associated with processing non-matching chunks. T_{OP} is thus the sum of the up-front operation cost (`OpCost()`) and the transfer cost (`XferCost()`). Operation cost is how long the disk plugin expects an operation to take. Transfer cost is how long the disk plugin expects the transfer of all chunks to *dsync* to take. $E[N_{delivered}]$ is the expected number of useful chunks found and is calculated by estimating the probability (p) of finding each required chunk when a particular disk operation is issued. $E[N_{delivered}]$ is then calculated as the sum of the probabilities across all source chunks.

⁴Determining an optimal schedule is NP-hard (Set Cover), and we've found that a greedy schedule works well in practice.

We present the details of computing the CPC for each disk operation in Section 6.2, along with the heuristics used to estimate the benefit of operations.

Metadata Groups

Schedule computation can involve thousands of disk operations and many thousands of chunks, with the corresponding scheduling decisions involving millions of *chunks * disk operations* pairs. Because the scheduler estimates the probability of finding a chunk using only metadata matching, all chunks with the same metadata have the same probabilities. Often, this means that all chunks from a particular source file are equivalent, unless those chunks appear in multiple source files. *dsync* aggregates these chunks in *metadata groups*, and performs scheduling at this higher granularity.

Metadata-group aggregation has two benefits. First, it greatly reduces the complexity and memory required for scheduling. Second, metadata groups facilitate making large sequential reads and writes by maintaining their associated chunks in request order. The receiver can then request the chunks in a group in an order that facilitates high-performance reads and writes.

5.3 Step B: Choosing Network Operations

After sorting the chunks by *CPC*, the scheduler selects which chunks to request from the network by balancing three issues:

- Request chunks that are unlikely to be found soon on disk (either because the operations that would find them are scheduled last or because the probability of finding the chunks is low).
- Give the network plugin enough chunks so that it can optimize their retrieval (by allocating chunks to sources using the rarest-random strategy [10]).
- Request chunks in order to allow both the sender and receiver to read and write in large sequential blocks.

The scheduler achieves this by computing the expected time of arrival (ETA) for each chunk C_j . If the chunk is found in the first operation, its ETA is simply the cost of the first operation. Otherwise, its cost must include the cost of the next operation, and so on, in descending *CPC* order through each operation:

$$\begin{aligned} T_i &= \text{Total cost of operation } i \\ \text{ETA}_j &= T_0 + (1 - p_{0,j}) \cdot T_1 + (1 - p_{0,j}) \cdot (1 - p_{1,j}) \cdot T_2 \\ &\quad + \cdots + \prod_i (1 - p_{i,j}) \cdot T_m \end{aligned}$$

where $p_{i,j}$ is the probability of finding chunk j using operation at position i .

The scheduler then prioritizes chunk requests to the network by sending those with the highest ETA first.

Operation Type Path	Op- Cost	T_{OP}	a.txt: p	b.txt: p	CPC
Time = 0					
STAT /d/new	1	9	1.000	1.000	1.125
Time = 1					
HASH /d/new/a.txt	4	4	1.000	0.350	0.741
STAT /d	1	9	0.444	0.444	2.533
HASH /d/new/core	64	64	0.007	0.007	1142.9
Expected Time to Arrival →			4	418.4	

Table 1: Disk operation optimization example.

5.4 Optimization Example

Table 1 shows *dsync*'s optimization for a sample transfer. The sender wants to transfer two 64 KB files, *a.txt* and *b.txt*, to a destination path on the receiver named */d/new*. The destination path has two files in it: *a.txt*, which is an exact copy of the file being sent; and a 1 MB *core* file. We assume the operation cost of STAT and CHUNK-HASH (cost for reading and hashing one chunk) to be 1 unit. At the first scheduling decision (Time = 0), the only available operation is a STAT of the target directory. The probability of locating useful chunks via this STAT operation is set to unity since we expect a good chance of finding useful content in the target directory of the transfer.

After performing this operation (Time = 1), the scheduler has the three listed options. The cost of the first option, HASHing */d/new/a.txt*, is 4 (OpCost(CHUNK_HASH) times the number of chunks in the file—64 KB / 16 KB chunk size). Because *a.txt* matches everything about the source file, *dsync* sets the probability of finding chunks from it to 1. We then compute the probability of finding chunks for file *b.txt* from the operation HASH(*/d/new/a.txt*) by comparing the metadata of files *a.txt* and source file *b.txt*; Both files belong to the same target directory, have the same size, but are only a file type match (both are text files), and do not have the same modification time. We give each of these factors a suitable weight to arrive at $p = 1 \cdot 1 \cdot 0.5 \cdot 0.7 = 0.350$ (Details in Section 6.2). The CPC for the operation is then computed as $T_{OP} = 4$ divided by $E[N_{delivered}](4 \cdot 1.0 + 4 \cdot 0.35)$. The costs for subsequent operations are computed similarly. The disk operations are then sorted as per their CPC. Intuitively, since hashing *a.txt* found on the disk appears more beneficial for the *.txt* files in the on-going transfer, its CPC is lower than the other operations.

Given the CPC ordering of operations, the expected time of arrival is then computed. Once again, because the probability of finding the chunks in *a.txt* during the first HASH operation (at Time = 0) is 1, the ETA for chunks in *a.txt* is simply equal to the cost of that first

operation. Thus, because the chunks in *b.txt* have a higher ETA from the disk, the scheduler will schedule those chunks to the network.

6 Implementation

This section provides details of (1) the software components and (2) the CPC parameters used in the current implementation of *dsync*.

6.1 Software Components

dsync is implemented using the Data-Oriented Transfer (DOT) service [21]. The *dsync_client* application transfers a file or file tree by first sending one or more files to its local DOT service daemon, receiving an object ID (OID) for each one. *dsync_client* creates a tree descriptor from these OIDs and the associated file metadata and sends the tree descriptor to the receiving *dsync_client*. The receiving *dsync_client* requests the OIDs from its local DOT daemon, which fetches the corresponding files using various DOT transfer plugins. Finally, *dsync_client* re-creates the file tree in the destination path.

The receiver side of *dsync* is implemented as a client binary and a DOT “transfer plugin”. DOT transfer plugins receive chunk requests from the main DOT system. The bulk of *dsync*'s functionality is a new DOT transfer plugin (called *dsync*), which contains the scheduler component that schedules chunk requests as described earlier, and a second plugin (*xdisk*) that searches for hashes in local files. For point-to-point transfers, *dsync* uses the DOT default transfer plugin that provides an RPC-based chunk request protocol. To properly size the number of outstanding requests, we modified this transfer plugin to send with each chunk the number of bytes required to saturate the network connection. This implementation requires kernel socket buffer auto-tuning, obtaining the queue size via `getsockopt`.

For peer-to-peer transfers, *dsync* uses DOT's existing Similarity-Enhanced Transfer (SET) network plugin [19]. SET discovers additional network peers using a centralized tracker. These network peers could be either seeding the entire file or could be downloading the file at the same time as the receiver (swarmers). To fetch chunks from swarmers more efficiently, SET exchanges bitmaps with the swarmers to learn of their available chunks. Finally, once it knows which chunks are available from which sources, SET requests chunks from these sources using the rarest-random strategy.

For *dsync* to work well on high-speed networks, we added a new zero-copy sending client interface mechanism to DOT to reduce data copies: `put(filename)`; clients using this mechanism must ensure that the file is not modified until the transfer has completed. DOT still stores additional copies of files at the receiver.

In addition to this interface, we improved DOT's efficiency in several ways, particularly with its handling of large (multi-gigabyte) files; as a result, our performance results differ somewhat from those described earlier [21].

6.2 Disk Operation CPC Parameters

For the implementation of the scheduling framework described in Section 5, we treat CACHE, HASH, and STAT operations individually.

6.2.1 CPC for CACHE Operations

There are two types of CACHE operations: CACHE-READ and CACHE-CHECK.

CPC_{CACHE-READ}: $\text{OpCost}(\text{CACHE-READ})$ involves the overhead in reading and hashing a single chunk. *dsync* tracks this per chunk cost ($\text{OpCost}(\text{CHUNK_HASH})$) across all disk operations; when operations complete, the scheduler updates the operation cost as an exponentially weighted moving average (EWMA) with parameter $\alpha = 0.2$, to provide a small amount of smoothing against transient delays but respond quickly to overall load changes.

CACHE-READ operations do not incur any transfer cost since the chunk is already in memory. Since one CACHE-READ operation results in a single chunk, the expected number of useful chunks delivered is optimistically set to 1.

CPC_{CACHE-CHECK}: $\text{OpCost}(\text{CACHE-CHECK})$ involves the overhead in looking up the chunk index. Since the chunk index is an in-memory hash table, we assume this cost to be negligible. Further, a CACHE-CHECK operation does not result in chunks; it results in more CACHE-READ operations. Thus, the ratio $\frac{\text{XferCost}(\cdot)}{E[N_{\text{delivered}}]}$ is set to the CPC of the best possible CACHE-READ operation. As seen above, this value equals $\text{OpCost}(\text{CHUNK_HASH})$.

6.2.2 CPC for HASH Operation

$\text{OpCost}(\text{HASH})$ is $\text{OpCost}(\text{CHUNK_HASH})$ multiplied by the number of chunks in the file. $\text{XferCost}(\text{HASH})$ is zero since the chunks are already in memory (they were just hashed). To calculate the expected number of useful chunks that can result from hashing a file, the scheduler calculates for each requested chunk, the probability that the chunk will be delivered by hashing this file (p).

The scheduler uses an extensible set of heuristics to estimate p based on the metadata and path similarity between candidate file to be HASHed and the source file that a requested chunk belongs to specified for the transfer. While these simple heuristics have been effective on our test cases, our emphasis is on creating a framework that can be extended with additional parameters rather than on determining absolute values for the parameters below. Note that we need only generate a good relative ordering

of candidate operations; determining the true probabilities for p is unnecessary. Currently, our heuristics use the parameters described below to compute p , using the formula:

$$p = p_{mc} \cdot p_{fs} \cdot p_{ss} \cdot p_{tm} \cdot p_{pc}$$

Max chunks: p_{mc} . If the candidate file is smaller than the source file, it can contain at most a $\frac{\text{size}_{\text{candidate}}}{\text{size}_{\text{source}}}$ fraction of the chunks from the source file; thus, any given chunk can be found with probability at most $p_{mc} = \frac{\min(\text{size}_{\text{candidate}}, \text{size}_{\text{source}})}{\text{size}_{\text{source}}}$.

Filename similarity: p_{fs} . An identical filename provides $p_{fs} = 1$ (i.e., this file is very likely to contain the chunk). A prefix or suffix match gives $p_{fs} = 0.7$, a file type match (using filename extensions) provides $p_{fs} = 0.5$, and no match sets $p_{fs} = 0.1$.

Size similarity: p_{ss} . As with filenames, an exact size match sets $p_{ss} = 1$. If the file size difference is within 10%, we set $0.5 \leq p_{ss} \leq 0.9$ as a linear function of size difference; otherwise, $p_{ss} = 0.1$.

Modification time match: p_{tm} . Modification time is a weak indicator of similarity, but a mismatch does not strongly indicate *dissimilarity*. We therefore set $p_{tm} = 1$ for an exact match, $0.9 \leq p_{tm} \leq 1$ for a fuzzy match within 10 seconds, and $p_{tm} = 0.7$ otherwise.

Path commonality: p_{pc} . Our final heuristic measures the directory structure distance between the source and candidate files. Intuitively, the source path `/u/bob/src/dsync` is more related to the candidate path `/u/bob/src/junk` than to `/u/carol/src/junk`, but the candidate path `/u/carol/src/dsync` is *more* likely to contain relevant files than either junk directory. The heuristic therefore categorizes matches as leftmost, rightmost, or middle matches.

The match coefficient of two paths is the ratio of the number of directory names in the longest match between the source and candidate paths over the maximum path length. How should p_{pc} vary with the length of the match? A leftmost or middle match reflects finding paths farther away in the directory hierarchy. The number of potential candidate files grows roughly exponentially with distance in the directory hierarchy, so a leftmost or middle match is assigned a p_{pc} that decreases exponentially with the match coefficient. A rightmost match, however, reflects that the directory structures have re-converged. We therefore assign their p_{pc} as the actual match coefficient.

Finally, if a chunk belongs to multiple source files, we compute its probability as the maximum probability computed using each file it belongs to (a conservative lower bound):

$$p = \max(p^{\text{file1}}, \dots, p^{\text{fileN}})$$

6.2.3 CPC for STAT Operation

The scheduler tracks $OpCost(STAT)$ similar to $OpCost(CHUNK_HASH)$. For STAT operations, the chunk probabilities are calculated using path commonality only, $p = p_{pc}$ and thus $E[N_{delivered}]$ can be computed. Once again, STATs do not find chunks; they produce more HASH operations. Thus, the ratio $\frac{XferCost()}{E[N_{delivered}]}$ for a STAT is set to the CPC of the *best* possible HASH operation that could be produced by that STAT. (In other words, the scheduler avoids STATs that could not possibly generate better HASH operations than those currently available).

Before the STAT is performed, the scheduler knows only the directory name and nothing about the files contained therein. The best possible HASH operation from a given STAT therefore would match perfectly on max chunks, file name, size, and modification time ($p_{mc} = p_{fs} = p_{ss} = p_{tm} = 1$). This lower bound cost is optionally weighted to account for the fact that most STATs do not produce a best-case HASH operation.

$$\begin{aligned} \text{best } p &= p_{pc} \cdot 1 \cdot 1 \cdot 1 \cdot 1 \\ CPC_{STAT} &= w \cdot \left(\frac{OpCost(STAT)}{\sum p_{pc}} + \frac{OpCost(CHUNK_HASH)}{\max(\text{best } p)} \right) \end{aligned}$$

7 Evaluation

In this section, we evaluate whether *dsync* achieves the goal set out in Section 2: Does *dsync* correctly and efficiently transfer files under a wide range of conditions? Our experiments show that it does:

- *dsync* effectively uses local and remote resources when initial states of the receivers are varied.
- *dsync* ensures that transfers are never slowed. *dsync*'s back-pressure mechanisms avoid overloading critical resources (a) when the network is fast, (b) when the disk is loaded, and (c) by not bottle-necking senders due to excessive outstanding requests.
- *dsync*'s heuristics quickly locate useful similar files in a real filesystem; *dsync* intelligently uses a pre-computed index when available.
- *dsync* successfully provides the best of both *rsync* and multi-source transfers, quickly synchronizing multiple machines that start with different versions of a target directory across 120–370 PlanetLab nodes using a real workload.

7.1 Method

Environment. We conduct our evaluation using Emulab [24] and PlanetLab [18]. Unless otherwise specified,

Emulab nodes are “pc3000” nodes with 3 GHz 64-bit Intel Xeon processors, 2 GB RAM, and two 146 GB 10,000 RPM SCSI hard disk drives running Fedora Core 6 Linux. We choose as many reachable nodes as possible for each PlanetLab experiment, with multiple machines per site. The majority of PlanetLab experiments used 371 receivers. The sender for the data and the peer tracker for the PlanetLab experiments are each quad-core 2 GHz Intel Xeon PCs with 8 GB RAM located at CMU.

Software. We compare *dsync*'s performance to *rsync* and CoBlitz [16] as a baseline. (We compare with CoBlitz because its performance already exceeds that of BitTorrent.) Achieving a fair comparison is complicated by the different ways these systems operate. *dsync* first computes the hash of the entire object before communicating with the receiver, while *rsync* pipelines this operation. We run *rsync* in server mode, where it sends data over a clear-text TCP connection to the receiver. By default, *dsync* uses SSH to establish its control channel, but then transfers the data over a separate TCP connection.

To provide a fair comparison, in all experiments, we measure the time *after* putting the object into *dsync* on the sender and establishing the SSH control channel, and we ensure that *rsync* is able to read the source file from buffer cache. This comparison is not perfect—it equalizes for disk read times, but *dsync* would normally pay a cost for hashing the object in advance. In addition *dsync* maintains a cache of received chunks, which puts extra pressure on the buffer cache. Hence, in most of our evaluation, we focus on the relative scaling between the two; in actual use the numbers would vary by a small amount.

To compare with CoBlitz, our clients contact a CoBlitz deployment across as many nodes as are available. (CoBlitz ran in a new, dedicated slice for this experiment.) We use CoBlitz in the out-of-order delivery mode. To illustrate *dsync*'s effectiveness, we also compare *dsync* to basic DOT, SET [19], *dsync*-1src and *dsync*-NoPressure when applicable. Basic DOT neither exploits local resources nor multiple receivers. SET is a BitTorrent-like swarming technique that does not use local resources. *dsync*-1src is simply *dsync* that does not use swarming. *dsync*-NoPressure is *dsync* configured to ignore back-pressure.

Finally, we start PlanetLab experiments using 50 parallel SSH connections. At most, this design takes two minutes to contact each node. All the experiment results are averaged over three runs. In the case of CoBlitz, the average is computed over the last three runs in a set of four runs to avoid cache misses. We log transfer throughput for each node, defined to be the ratio of the total bytes transferred to the transfer time taken by that node.

Scenarios. In many of our experiments, we configure the local filesystem in one of the following ways:

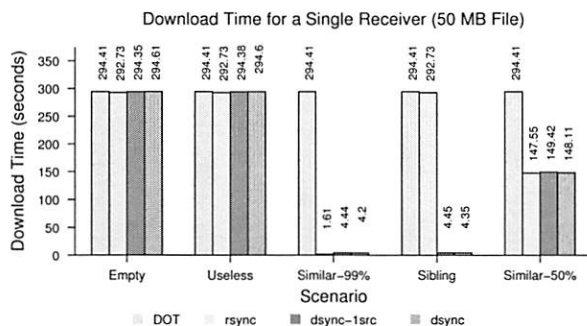


Figure 2: Transfer time for a single receiver using five initial filesystem configurations over a 1.544 Mbps link.

- Empty: The destination filesystem is empty, affording no opportunities to find similar files.
- Useless: The destination filesystem has many files, but none similar to the desired file. The filesystem is large enough that *dsync* does not completely explore it during a transfer.
- Similar: The destination filesystem contains a file by the same name as the one being transferred in the final destination directory. We set the similarity of this file with the file being transferred to either 50% or 99%.
- Sibling: The destination filesystem contains a 99% similar file with the same name but in a directory that is a sibling to the final destination directory.

7.2 *dsync* works in many conditions

In this section, we explore *dsync*'s performance under a variety of transfer scenarios, varying the number of receivers, the network conditions, and the initial filesystem configurations.

7.2.1 Single receiver

We first use a simple 50 MB data transfer from one source to one receiver to examine whether *dsync* correctly finds and uses similar files to the one being transferred and to evaluate its overhead relative to standard programs such as *rsync*. The two nodes are connected via a 1.544 Mbps (T1) network with 40 ms round-trip delay. Figure 2 compares the transfer times using the original DOT system, *dsync-1src*, *dsync*, and *rsync* under the five filesystem scenarios outlined above. This figure shows three important results:

***dsync* rapidly locates similar files.** In the “Similar-99%” configuration, both *dsync* and *rsync* locate the nearly-identical file by the same name and use it to rapidly complete the data transfer. In the “Sibling” configuration, only *dsync*⁵ locates this file using its search heuristics.

⁵*rsync* supports a manual Sibling mode where the user must point *rsync* at the relevant directory.

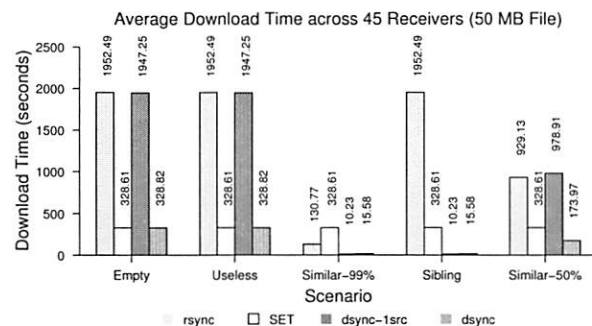


Figure 3: Transfer time across 45 nodes using five filesystem configurations over 1.544 Mbps links.

***dsync*'s performance is as good as its subsystems individually.** When transferring over the network, *dsync* is as fast as the DOT system on which it builds. *dsync* incurs negligible penalty over *dsync-1src* when using both disk and swarming plugins.

***dsync* adds only small overhead vs. *rsync*.** In the similar-99 case, *dsync* requires 2.8 seconds more transfer time than *rsync*. The overhead over the basic time to read and chunk (2.3 seconds) comes from a mix of the DOT system requesting chunk hashes *before* reading from either the network or the local filesystem (0.4 seconds), *dsync*'s post-hash index creation (0.3 seconds) and an artifact of the Linux kernel sizing its TCP buffers too large, resulting in an excess queue for *dsync* to drain before it can fetch the last chunk from the single source (250 KB at 1.5 Mbps = 1.4 seconds).

7.2.2 Multiple receivers, homogeneous initial states

We next explore *dsync*'s ability to swarm among multiple receivers. In this Emulab experiment, we use a single 10 Mbps sender and 45 receivers with T1 links downloading a 50MB file. All the receivers have homogeneous initial filesystem configurations. The receivers use a mix of hardware, ranging from pc650 to pc3000. The sender is a pc850. Figure 3 shows the average and standard deviation of the transfer times for *rsync*, SET, *dsync-1src* and *dsync* as the initial configurations are varied.

***dsync* effectively uses peers.** The “Empty” and the “Useless” scenarios show the benefit from swarming – In *rsync* and *dsync-1src*, all the receivers obtain the data from a single sender, dividing its 10 Mbps among themselves. *dsync* (and SET) swarm among the receivers to almost saturate their download capacities, resulting in 6× speedup.

***dsync* effectively combines peering and local resources.** In the “Similar-99%” scenario, *dsync* is 21× faster than SET because it uses the local file. *dsync* is also 8× faster than *rsync* in this scenario, because the single *rsync* sender becomes a computational bottleneck.

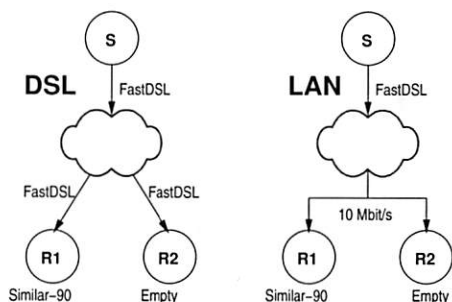


Figure 4: Network and initial states of R1 and R2.

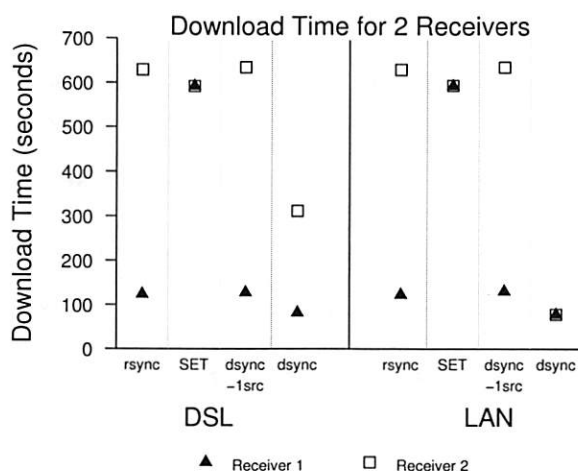


Figure 5: Transfer time in a two-receiver transfer.

Finally, in the “Similar-50%” scenario, all schemes except SET effectively use the local data to double their throughput.

7.2.3 Multiple receivers, heterogeneous initial states

This experiment shows how *dsync* can synchronize multiple machines whose disks start in different initial configurations from each other by intelligently using a combination of similar local files, the original sender, and the other recipients. Figure 4 depicts two scenarios, and Figure 5 shows the download time for a 50 MB file with *rsync*, SET, *dsync-1src*, and *dsync*.

In both *rsync* and *dsync-1src*, Receiver 1 uses the similar file on its local disk to speed the transfer while Receiver 2 fetches the entire file from the source. Because the sender is the bottleneck in both cases, the DSL and LAN cases perform similarly. SET alone behaves like other peer-to-peer swarming systems; Receiver 1 does not use the local file. Instead, the two receivers collectively fetch one copy of all of the file data from the sender. Receiver 2’s download is faster with SET than with *rsync* because with *rsync*, the clients fetch 1.10 copies (100% and 10%, respectively) of the data; SET’s rarest-first heuristic ensures that the receivers do not request the same chunks.

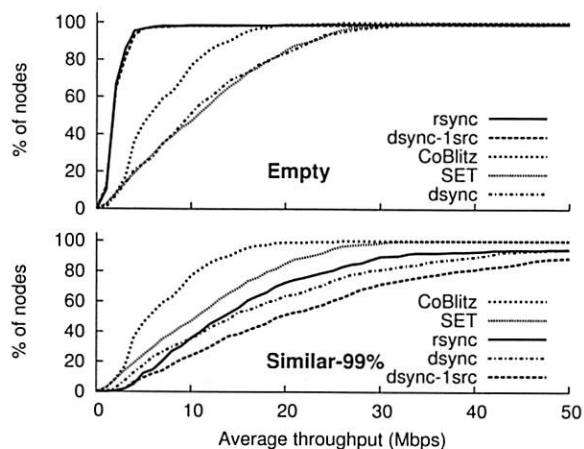


Figure 6: CDF of average throughput across 371 PlanetLab nodes for different filesystem configurations.

dsync shows drastic improvement from searching the local disk and downloading from a peer. Receiver 1 fetches most of the file from the local disk and offers it to Receiver 2. In the DSL scenario, Receiver 2 fetches the file from both the sender and Receiver 1, transferring the file about twice as fast as *rsync* can. In the LAN scenario, Receiver 2 grabs almost all of the bytes over its fast link to Receiver 1, and the two receivers fetch only the missing 10% of the data from the slow source. Overall, *dsync* automatically uses the best resources to download the file, using the disk when possible and transferring only one copy of the remaining data over the network.

7.2.4 Multiple receivers in the wild (PlanetLab)

We now explore the performance of *rsync*, CoBlitz, SET, *dsync-1src* and *dsync* when all available PlanetLab nodes simultaneously download a 50 MB file. All the nodes are in identical initial filesystem configurations.

Figure 6 shows the throughput for all the schemes in the “Empty” scenario. The single sender becomes a bottleneck in *rsync* and *dsync-1src*. CoBlitz uses multi-source downloads and caching to outperform *rsync*. Somewhat unexpectedly, *dsync* outperforms CoBlitz; the median *dsync* node is $1.67\times$ faster. The most evident reason for this is that *dsync* receivers make more aggressive use of the original source, drawing 1.5 GB from the sender vs. CoBlitz’s 135 MB. Thus, CoBlitz is more sender friendly. However, CoBlitz’s performance is impaired as the remaining senders are extremely busy PlanetLab nodes. *dsync*’s load balancing makes more effective (but perhaps less friendly) use of the fast source. *dsync*’s performance is again comparable to that of the underlying SET transfer mechanism in the “Empty” scenario: suggesting that *dsync*’s filesystem searching does not slow down the transfers even on busy nodes.

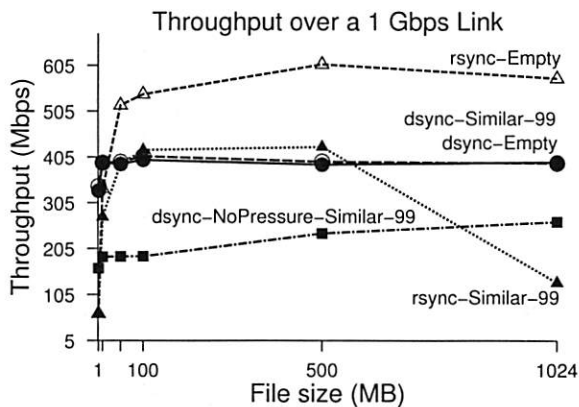


Figure 7: Throughput over a 1 Gbps link vs. file size.

In the “Similar-99%” scenario, *dsync* automatically uses local resources when appropriate to improve the median throughput by 50% over “Empty” scenario. An interesting note about this figure is that *dsync*-lsrc (no swarming) is slightly *faster*, because the additional network load appeared to slow down the PlanetLab nodes. This effect did not appear in similar tests on Emulab; we accept this as a case in which *dsync*’s generality made it perform slightly worse in one particularly extreme example. *dsync* did, however, correctly reduce network traffic in these cases.

7.3 *dsync*’s back-pressure is effective

This section shows that *dsync* effectively uses its back-pressure mechanisms to adapt to overloaded critical resources in order to retain high performance.

7.3.1 Gigabit network

In this experiment, we transfer files varying from 1 MB to 1 GB over a 1 Gbps network between two nodes. This network is faster than the disk, which can sustain only 70 MB/sec, or 550 Mbps. Thus, the disk is the overloaded critical resource that limits transfer speed. Figure 7 shows throughput as file size increases for the various systems.

***dsync* uses back-pressure to defer disk operations.** *dsync* correctly notices that the network is faster than the disk and ignores the local file. Thus, *dsync*-Similar-99% and *dsync*-Empty perform similarly. *rsync*-Similar-99%, however, suffers significantly when it tries to use a 1 GB local file because it competes with itself trying to write out the data read from the disk.

It is noteworthy that *dsync* outperforms *rsync*-similar even though the prototype *dsync* implementation’s overall throughput is lower than *rsync*-Empty as a result of additional data copies and hashing operations in the underlying DOT system. This figure also shows that *dsync*’s improvements come mostly from its back-pressure mechanisms and not simply from using both the network and

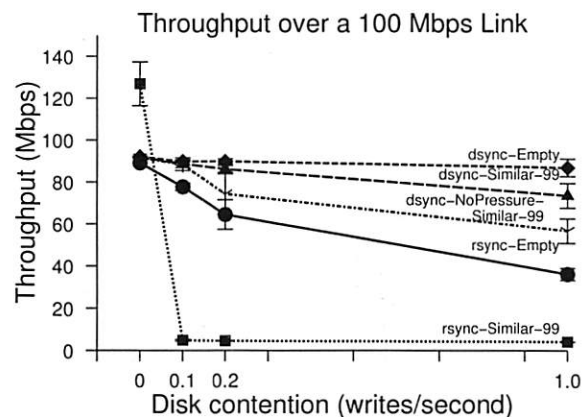


Figure 8: Throughput vs. disk contention for a 1 GB file.

the disk. While *dsync*-NoPressure also slightly outperforms *rsync* in Similar-99% case, its performance is much lower than that of the pressure-aware *dsync*.

7.3.2 Slow disk

The gigabit case is not a rare exception to be handled by an “if” statement. Critical resources can be overloaded in a variety of scenarios. For example, we simulate slow disk scenarios using a process performing bulk writes (1 MB at a time) to disk and varying the frequency of writes. The single sender and the receiver are connected by 100 Mbps LAN and transfer a 1 GB file. Figure 8 shows the throughput as disk load increases.

***dsync* again uses back-pressure to defer disk operations.** *rsync*-Similar-99%, however, still tries to read from the overloaded local disk, and its performance degrades much more rapidly than fetching from the network alone.

7.3.3 Dynamic adaptation to changing load

Figure 9 shows a transfer in the Slow Disk scenario (above), in which we introduce disk load 20 seconds into the transfer. Each plot shows the number of bits/second read from each resource during the transfer.

***dsync* dynamically adapts to changing load conditions among resources.** We observe that *dsync* backs off from reading from the disk when the contention starts. This helps *dsync* to finish writing the transferred file more quickly (101 seconds) than *dsync*-NoPressure (117 seconds).

7.4 *dsync* effectively uses the local disk

This section highlights *dsync*’s ability to discover useful data on disk regardless of index state. *dsync* uses a pre-computed index when available, and its heuristics for data discovery otherwise.

***dsync* correctly decides between index reads and sequential reads.** To evaluate *dsync*’s use of the index, we transfer files between 10 MB and 1 GB between two

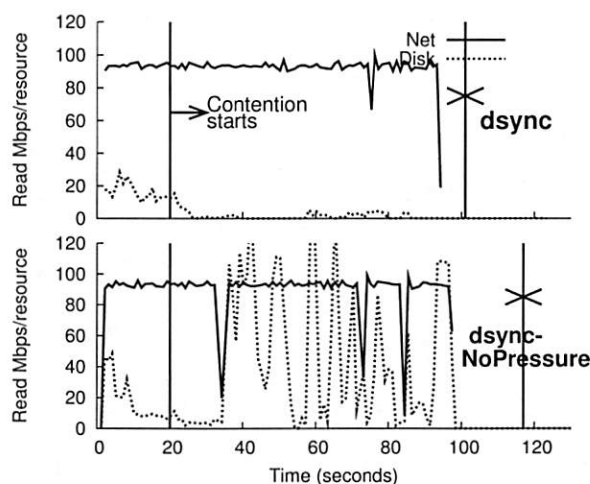


Figure 9: Throughput as disk contention starts 20 seconds into transfer. X indicates transfer completion.

	10	50	100	1024
<i>dsync</i>	0.94	4.39	7.75	167
<i>dsync-naive</i>	0.91	4.10	8.63	525

Table 2: Transfer time (seconds) from intelligent use of proactive index as file size is varied from 10 MB to 1 GB.

nodes. The receiver maintains a pre-computed index of the files on the disk, and all files being transferred are available on the receiver's disk. *dsync-naive* satisfies the cache hit for each chunk request by opening the relevant file, seeking to the chunk offset and reading the required bytes. *dsync*, however, observes that several chunks are desired from the same file and sequentially reads the file to serve all cache hits. The results of this test are shown in Table 2. As the file size increases, this strategy is more effective—a 1 GB file is copied 3× faster.

dsync's filesystem search heuristics are effective at exploring the disk to identify useful chunks. In this experiment, the sender wants to transfer a file called “a.out” into the destination path “/s/y/n/c/test/” on the receiver. Each set of rows in Table 3 represents iterations of the experiment with the useful file in a different directory. We measure the number of STAT and HASH operations and the number of megabytes hashed up to and including when *dsync* hashed the useful file. We disabled fetching chunks from the network since our intent was to measure the discovery process of the heuristics and not transfer time. We compare our heuristics with two simple strategies: (a) An allstats strategy where we “stat” the entire hard disk and hash the files in the descending order of their usefulness, and (b) a breadth-first strategy (BFS) that traverses the directory hierarchy by hashing all files in its sub-tree before going to its parent.

The root of the filesystem is the desktop machine home directory of one of the authors. The file system had 445,355 files in 37,157 directories, occupying 76 GB. The alternate paths we explore represent a few feasible locations in which a duplicate copy of a file might exist in a home directory. We do not claim that this set is representative of other filesystems, but it demonstrates that the heuristics perform well in these (reasonable) scenarios. We performed no tuning of the heuristics specific to this filesystem, and never used it as a test during development.

In the top group, the useful file is in the destination directory. Here, the heuristics find the file quickly and do not hash any other files beforehand. When the useful file has a completely different name “diff”, *dsync* issues STAT requests of several other directories before hashing it. In the second group, the useful file is in a sibling directory to the destination path. Here, *dsync's* heuristics perform several additional STATs before finding the useful file, but relatively few additional hashes. In fact, the scheduler suggests only one extraneous HASH operation for a file with size match. BFS performs reasonably well in scenarios 1-6. Allstats incurs a constant 56.17 seconds to stat the entire filesystem before hashing the useful file.

The final group in the table shows the results when the useful file is in a completely unrelated directory on the receiver's file system. Here, when the file name matches exactly, *dsync* STATs almost 15% of the directories in the file system, but only hashes two extraneous files. The last line of the table reveals an interesting result: when the useful file has neither a name or path match, *dsync* inventoried the entire file system looking for useful files, but still only hashed 175 MB of unnecessary data. This results in performance similar to allstats. BFS, however, degrades as it hashes ~4 GB of data. Thus, *dsync* intelligently adapts to different filesystem configurations unlike simple strategies.

7.5 Real workload

dsync substantially improves throughput using a workload obtained from our day-to-day use—running our experiments required us to frequently update PlanetLab with the latest *dsync* code. This workload consisted of 3 binaries (totaling 27 MB): the DOT daemon (gtcd), a client (gcp) and the disk read process (aiod). We choose one such snapshot during our experimentation where we synchronized 120 PlanetLab nodes. A third of these nodes did not have any copy of our code, another third had a debug copy in a sibling directory and the last third had almost the latest version in the destination directory. We also repeated this experiment for 371 nodes.

Figure 10 shows the CDF of average throughput across these nodes for *rsync*, SET and *dsync*. Using *dsync*, the median node synchronizes 1.4-1.5× faster than it does using SET, and 5× faster than using *rsync*.

Path of Useful File	Size, Similarity (MB, %)	<i>dsync</i>			<i>BFS</i>		
		STAT Ops	Files (MB) Hashed	Seconds	STAT Ops	Files (MB) Hashed	Seconds
1. /s/y/n/c/test/a.out	12.0, 100%	1 (0%)	1 (12)	0.30	1 (0%)	8 (70.05)	1.25
2. /s/y/n/c/test/a.out	12.1, 11%	1 (0%)	1 (12.1)	0.29	1 (0%)	8 (70.05)	1.25
3. /s/y/n/c/test/diff	12.0, 100%	373 (1%)	1 (12)	0.76	1 (0%)	8 (70.05)	1.25
4. /s/y-old/n/c/test/a.out	12.0, 100%	879 (2.37%)	2 (24.03)	1.84	364 (0.97%)	3212 (139.4)	4.17
5. /s/y-old/n/c/test/a.out	12.1, 11%	879 (2.37%)	2 (24.03)	1.84	364 (0.97%)	3212 (139.4)	4.17
6. /s/y-old/n/c/test/diff	12.0, 100%	2225 (5.98%)	2 (24.03)	4.67	364 (0.97%)	3212 (139.4)	4.17
7. /a.out	12.0, 100%	6038 (16.25%)	3 (34.03)	8.90	6962 (18.74%)	51,128 (4002.4)	1771.9
8. /a.out	12.1, 11%	6038 (16.25%)	3 (34.03)	8.90	6962 (18.74%)	51,128 (4002.4)	1771.9
9. /diff	12.0, 100%	37,157(100%)	227 (173.8)	68.3	6962 (18.74%)	51,128 (4002.4)	1771.9

Table 3: Performance of *dsync* heuristics on a real directory tree. Files with 11% similarity are an old version of “a.out”.

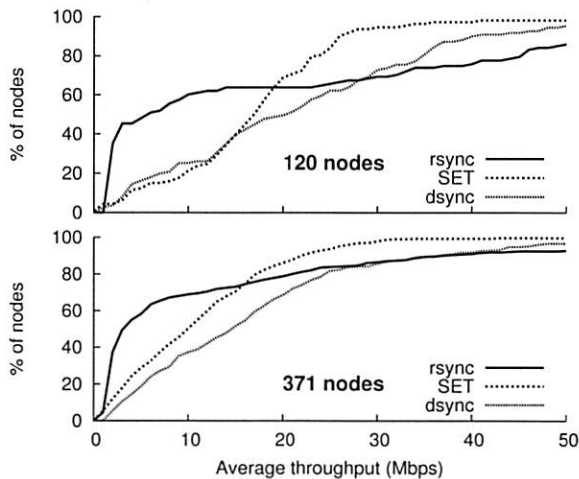


Figure 10: CDF of average throughput when *dsync* is used for software distribution on Planetlab.

8 Related Work

Our work fits into the broad scope of systems that leverage an excess of one resource to compensate for the lack of another. Examples range from the familiar use of caching to avoid computation to more contemporary systems that use spare CPU cycles, in the form of speculative execution, to hide disk latency [3] and network latency [15]; or that mask network latency or limited capacity by explicit caching or prefetching [9]. *dsync* specifically uses local computation and storage resources to replace network bandwidth, when such a trade-off is appropriate.

Like many such systems, ensuring that *dsync* does not create a scarcity of a local resource is key to performing well under a wide range of conditions. These problems do not just occur with disk contention; for example, the benefits of Web prefetching can be eclipsed if the prefetching traffic slows down foreground traffic, a situation that can again be solved by strictly prioritizing traffic that is guaranteed to be useful [9].

Many systems address the problems of drawing from multiple sources that have heterogeneous performance. The Blue File System [14], for example, selects sources

from a group of storage devices based on power and performance characteristics. The River cluster filesystem [2] shares our goal of automatically using sources of different rates (though in its case, different disk rates within a cluster). Like *dsync*, River does so by managing the request queue to each source. Exposing the queues between components helps systems such as SEDA shed load intelligently under overload conditions [23], much as *dsync*’s queue monitoring avoids local resource contention.

As we noted in the introduction, many of the techniques that *dsync* uses to obtain chunks are borrowed from systems such as *rsync* [22], LBFS [13], CFS [6], and Shark [1]. A particularly noteworthy relative is Shotgun, which used the Bullet’ content distribution mesh to disseminate the diff produced by *rsync* batch mode [11]. This approach works well—it avoids the overhead of multiple filesystem traversals on the sender and greatly reduces the amount of network bandwidth used—but *rsync* batch mode requires that the recipients be in identical starting states and is subject to the same limitations as *rsync* in finding only a single file to draw from.

dsync uses a peer-to-peer content distribution system to allow nodes to swarm with each other. For implementation convenience, we used the SET plugin from DOT, but we did not use any of its unique features—our system could use any transfer system that allows transferring chunks independent of a file. In keeping with our philosophy of opportunistic resource use and broad applicability, the most likely alternative distribution system for *dsync* is CoBlitz [16] since it is cache-based and does not require that sources “push” data into a distribution system.

Finally, several forms of system support for chunk-based indexing provide possible opportunities for improving the timeliness and efficiency of the pre-computed index. Linux’s `inotify()` would at least permit an index daemon to re-hash files when they change. A content-addressable filesystem would avoid the need for an external index at all, provided its chunking and hashing was *dsync*-friendly. The CZIP [17] approach of adding user-level chunk index headers to files would permit *dsync* to

much more rapidly examine new candidate files during its filesystem exploration.

9 Conclusion

dsync is a data transfer system that correctly and efficiently transfers files under a wide range of operating conditions. *dsync* effectively uses all available resources to improve data transfer performance using a novel optimization framework. With an extremely fast sender and network, *dsync* transfers all data over the network instead of transferring the data more slowly from the receiver's local disk; with a slow network, it will aggressively search the receiver's disk for even a few chunks of useful data; and when sending to multiple receivers, *dsync* will use peer receivers as additional data sources.

While combining network, disk-search, and peer-to-peer techniques is conceptually simple, doing so while ensuring that resource contention does not impair performance is difficult. The keys to *dsync*'s success include adaptively deciding *whether* to search the local disk, intelligently scheduling disk search operations and network chunk requests to minimize the total transfer time, and constantly monitoring local resource contention. Making good scheduling decisions also requires that *dsync* deal with practical system issues such as ensuring large sequential read/write operations, that can be hidden behind the content-based naming abstraction.

Our evaluation shows that *dsync* performs well in a wide range of operating environments, achieving performance near that of existing tools on the workloads for which they were designed, while drastically outperforming them in scenarios beyond their design parameters.

Acknowledgments

We thank the anonymous reviewers who provided valuable feedback on this work; Shimin Chen for discussion about database optimization strategies; Phil Gibbons and Anupam Gupta for discussions about the set cover problem; and KyoungSoo Park and Vivek Pai for their help running CoBlitz. This work was supported in part by NSF CAREER award CNS-0546551 and DARPA Grant HR0011-07-1-0025.

References

- [1] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX NSDI*, May 2005.
- [2] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with river: Making the fast case common. In *Proc. IOPADS: Input/Output for Parallel and Distributed Systems*, May 1999.
- [3] F. Chang and G. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. 3rd USENIX OSDI*, Feb. 1999.
- [4] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, June 2003.
- [5] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th USENIX OSDI*, Dec. 2002.
- [6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [7] T. E. Denehy and W. W. Hsu. Duplicate Management for Reference Data. Research Report RJ10305, IBM, Oct. 2003.
- [8] F. Dougliis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
- [9] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin. NPS: A non-interfering deployable web prefetching system. In *Proc. 4th USENIX USITS*, Mar. 2003.
- [10] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. 19th ACM SOSP*, Oct. 2003.
- [11] D. Kostic, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proc. USENIX Annual Technical Conference*, Apr. 2005.
- [12] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proc. 1st USENIX NSDI*, Mar. 2004.
- [13] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [14] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proc. 6th USENIX OSDI*, pages 363–378, Dec. 2004.
- [15] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.
- [16] K. Park and V. Pai. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd USENIX NSDI*, May 2006.
- [17] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting practical content-addressable caching with CZIP compression. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.
- [18] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. 1st ACM Hotnets*, Oct. 2002.
- [19] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. 4th USENIX NSDI*, Apr. 2007.
- [20] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [21] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd USENIX NSDI*, May 2006.
- [22] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.
- [23] M. Welsh, D. Culler, and E. Brewer. SED: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Dec. 2002.

Handling Flash Crowds from your Garage

Jeremy Elson and Jon Howell

Microsoft Research

{jelson,howell}@microsoft.com

Abstract

The garage innovator creates new web applications which may rocket to popular success – or sink when the flash crowd that arrives melts the web server. In the web context, *utility computing* provides a path by which the innovator can, with minimal capital, prepare for overwhelming popularity. Many components required for web computing have recently become available as utilities.

We analyze the design space of building a load-balanced system in the context of garage innovation. We present six experiments that inform this analysis by highlighting limitations of each approach. We report our experience with three services we deployed in “garage” style, and with the flash crowds that each drew.

1 Introduction

For years, developers and researchers have refined ways of scaling Internet services to larger and larger capacities. Many well-known techniques are now available [33, 17, 4, 19, 6, 36, 15, 13].

But capacity is expensive: racks full of web servers, database replicas, and load balancing switches require a significant up-front investment. For popular sites, where load is consistently high, it is easy to justify this investment; the resources will not be idle. Less popular sites can not justify the expense of large idle capacity. But how can an unpopular site make the transition to popular—when “flash crowds” often make that transition almost instantaneous and without warning?

In this paper, we consider the question of scaling through the eyes of a character we call the *garage innovator*. The garage innovator is creative, technically savvy, and ambitious. She has a great idea for the Next Big Thing on the web and implements it using some spare servers sitting out in the garage. The service is up and running, draws new visitors from time to time, and makes some meager income from advertising and subscriptions. Someday, perhaps, her site will hit the jackpot. Maybe it will reach the front page of Slashdot or Digg; maybe Valleyway or the New York Times will mention it.

Our innovator may get only one shot at widespread publicity. If and when that happens, tens of thousands of people will visit her site. Since her idea is so novel,

many will become revenue-generating customers and refer friends. But a flash crowd is notoriously fickle; the outcome won't be nearly as idyllic if the site crashes under its load. Many people won't bother to return if the site doesn't work the first time. Still, it is hard to justify paying tens of thousands of dollars for resources *just in case* the site experiences a sudden load spike. Flash crowds are both the garage innovator's bane and her goal.

One way out of this conundrum has been enabled by contemporary *utility computing*. More and more of the basic building blocks of scalability—network bandwidth, large-scale storage, and compute servers—are now available in forms analogous to traditional utilities like electricity and water. That is, a contract with a utility has very little overhead, gives you access to vast resources almost instantly, and only bills you for the resources you use.

Over the past year, our research group created three web sites that experienced sudden surges in popularity, including one that was (literally) Slashdotted. Each was implemented using publicly available computing utilities and was able to withstand its flash crowd at low cost.

1.1 Contributions

In this paper, we contribute a detailed analysis of the issues and tradeoffs a typical garage innovator will encounter when building low-cost, scalable Internet services. We explain these tradeoffs in Section 3. Our analysis draws heavily from both a series of controlled micro-benchmark experiments, which we describe in Section 4, and wisdom gleaned from our own deployment of three “garage-scalable” services, all of which were subject to real flash crowds. These case studies, found in Section 5, report our design and implementation decisions, how each service responded to a flash crowd, and lessons learned from each experience.

Before diving into our analysis, however, we first lay groundwork in Section 2, which describes the current state of utility computing and briefly reviews the most common strategies for building scalable Internet services.

2 Contemporary Utility Computing

The past two years have seen a surge of tools that are wonderfully useful for garage innovators. We describe several of them in this section. First, we offer a list of what we think are the properties essential for garage use:

Low overhead during the lean times. Cost should be proportional to use, not to capacity. During long periods of unpopularity, a garage budget can't pay for the huge capacity that might someday be needed.

Highly scalable. The garage-based service may only need one server today, but when the flash crowd comes, it might need 20 or 200. Worst-case flash crowd resources have to be available: if a service is underprovisioned, there is no point in using it.

Quickly scalable. It's not enough that large-scale resources are *available*; they have to be available *quickly*. There's no time to call customer service, ask for an upgraded account, and start configuring machines. Flash crowds are notoriously fickle. If a service can't scale in near-immediate response to a surge of interest, there is no point in using it.

Services that meet these criteria are often referred to as *utility computing*, a term coined by John McCarthy in 1961. Utility computing services rely on *statistical multiplexing*: providing service to a large number of customers whose load spikes are likely to be de-correlated.

An illustrative shift towards utility computing can be found in the way large colocation centers sell bandwidth to customers. It is common today to see it billed as a utility: a customer gets a fast (say, 100Mbps) connection to her servers. The entire 100Mbps is usable, and actual usage is metered. Using very little bandwidth costs very little; a sudden usage surge is billed accordingly. This contrasts with circuits installed to an individual customer's site, virtually all of which are billed according to peak capacity regardless of actual usage.

2.1 Building Blocks

In this section, we lay the foundation for the rest of the paper, describing some of the utility computing services that have arisen in the past few years. Then, in Section 2.2, we describe a few well-known scaling architectures and describe how a garage innovator can implement them using the utility building blocks that are available today.

2.1.1 Storage Delivery Networks

One great boon to the garage innovator has been the rise of Storage Delivery Networks (SDNs), such as Amazon's S3 [26] and the Nirvanix platform [25]. SDNs have interfaces that resemble a simple managed web server. Developers can upload static files such as web pages and images that the SDN will store and serve to clients using standard HTTP.

Unlike traditional managed web hosting, often implemented using a single machine, SDNs are large clusters of tightly coupled machines. The implementation details of data replication, distributed consensus, and load distribution are all hidden behind the simple static-content interface. A single customer's flash crowd can potentially use the full power of the entire cluster.

This strategy should sound familiar: SDNs are similar to Content Distribution Networks (CDNs) such as Akamai [1] and Limelight Networks [21]. CDNs and SDNs have a number of technical differences; for example, SDNs are typically located in a single datacenter, while CDNs minimize latency using thousands of Internet points of presence. CDNs are far less attractive to garage innovators than SDNs, however—not for technical reasons, but economic ones. The cost of entry into a CDNs is typically high, as well as frustratingly opaque [29] (“contact a sales representative for more information!”). Large start-up costs and minimum bandwidth commitments place most CDNs out of the reach of garage innovators who don't yet have the budget associated with a wide audience. S3 and Nirvanix have no start-up or recurring costs; they are strictly fee-for-service utilities. A customer who serves one gigabyte of data in a month will literally be billed 20 cents for the month. There may be no fundamental technical or economic reason why CDNs cannot adopt a similar billing model; perhaps some day they will.

SDNs thus fill a useful niche today for the garage innovator. They are more quickly scalable than typical managed hosting servers; they do not carry the significant expense of geo-distribution that is inherent to CDNs; and their pricing models allow flash crowd-capable content distribution without any significant investment until the flash crowd arrives.

2.1.2 Commodity Virtualization

We mentioned in the introduction of Section 2 that colocation centers now charge for bandwidth in a utility-computing, garage-friendly way. However, up until recently, the only way to exploit utility bandwidth was to buy or rent a server and pay a monthly fee (typically a few hundred dollars) to host it. For the garage innovator on a truly shoe-string budget, this can be prohibitive. Dozens of hosting companies now offer virtual machines for rent, typically starting at around 20 dollars a month. There are dozens of examples (search the Internet for “virtual private servers”). Setup usually takes about a day. Developers can use these virtual machines to exploit the pay-per-use bandwidth of a colo facility without the overhead of using an entire physical machine.

Garage innovators can also exploit the fact that virtual servers are so widely available around the globe—offered by hosting providers in dozens of locations

around the United States, Europe, Asia, and the Pacific Rim. By renting several of them, a developer on a limited budget can achieve a level of geographic diversity that was formerly only possible for large-scale Internet services with large-scale budgets.

2.1.3 Compute Clouds

While the widespread availability of virtual servers has been a boon, it did have an important limitation. The flexibility of a virtual server was typically only in burstable bandwidth. If an application is CPU or disk intensive, a flash crowd doesn't just need more bandwidth, it needs more servers. Consequently, in the past year, companies have begun to follow the utility computing model for entire virtual machines, not just the bandwidth they consume.

Amazon's EC2 "elastic compute cloud" [2] and FlexiScale [35] stand out in this area. They allow developers to configure and save a virtual machine image, then create multiple running instances of that image. Images can be instantiated in about one minute. Usage is billed by the hour, rather than by the month. Virtual machines can be started and stopped using a simple programmatic API. This makes it possible for a garage innovator to create an image of a machine running her Internet service, monitor the load, and almost instantly increase the number of running instances if needed. As we will see in Section 2.2, there are several ways of scalably distributing the load of an Internet service over multiple servers, each with different advantages.

2.1.4 DNS Outsourcing

Another useful computing utility is outsourced Domain Name System (DNS) hosting. DNS traffic usually accounts for a small part of a site's resource budget, but outsourcing DNS is useful because it prevents DNS from becoming a single point of failure for garage-based services. (We will explore this further in Section 3.5.)

Typical services in this space are the highly redundant and low-cost UltraDNS [24] and Granite Canyon's free Public DNS Service [14]. Their DNS servers replicate and serve the DNS for customer domains. They automatically copy the authoritative DNS configuration every time they receive a change notification.

2.1.5 A missing piece: relational databases

Dynamic web services are often implemented as a collection of stateless front-end servers that paint a user interface over data stored in a relational database back end. Relational databases' powerful transactional model provides idiot-proof concurrency management, and their automatic indexing and query planning relieve programmers from the burden of designing efficient persistent data structures [8]. However, implementing highly-

scalable databases that retain the full generality of the relational model has proven elusive. Scalable databases typically abandon full transactionality, arbitrary queries, or both. Utility access to a scalable database is therefore even further in the future.

There exist lightweight scalable utility databases, such as S3 and Amazon's SimpleDB [3]. Later in the paper (Sections 5.2 and 5.3), we describe experiences substituting the conventional relational database, sometimes with a lightweight database, and other times with alternate workarounds. Every approach incurs a higher development cost over using a powerful relational database, a cost of scalability we do not know how to eliminate today.

2.2 Scaling Architectures

Before proceeding to our analysis (Section 3), we will briefly review some of the common scaling architectures used today for Internet services. This discussion focuses purely on the Internet-facing part of the system: that is, methods for rendezvous of a large number of Internet clients to the large number of Internet-facing servers that handle their sessions. We do not consider scalability of back-end elements such as databases. (Our case studies in Section 5 will revisit the scalable back-end issue.)

This section will describe each design, and briefly touch on its main advantages and disadvantages.

2.2.1 Using the bare SDN

Most of the design alternatives we will consider assume that an innovator's web site has dynamic content, and therefore requires a compute cluster that can run garage code. However, some web sites rely heavily (or even exclusively) on static content. For example, video sharing sites typically show dynamic web pages that display the latest comments and ratings, but the contained video is a much larger static object.

In these cases, simply storing the static parts of a web site on an SDN is near ideal for the garage innovator. Using the SDN costs our innovator virtually nothing upfront beyond the small fee for storage. Because of the statistical multiplexing we described in Section 2.1.1, the SDN is likely to be highly available even during the arrival of a flash crowd.

The main disadvantage of using an SDN, of course, is that it serves only static content.

2.2.2 DNS load-balanced cluster

We now turn our attention to clusters running custom code designed by our garage innovator. One simple approach is to use the DNS protocol to balance load: A collection of servers implement the custom web service, and a DNS server maps a single name to the complete list of IP addresses of the servers [5]. Standard DNS

servers will permute this list in round-robin fashion; if clients try the addresses in order, then various clients will be randomly spread across the servers. Clients should also fail over to a second address if the first one does not reply, affording fault tolerance. (In sections 4.3 and 4.4, we will show how these properties can fail.)

When a flash crowd arrives, new servers are brought online, and the DNS record is updated to include their addresses. Clients of ISPs that have cached the previous record won't see the new servers until the old record expires. Fortunately, the nature of a flash crowd means that most of the traffic is new. On the other hand, record expiration does reduce the responsiveness of DNS load balancing to server failure.

A startup called RightScale offers a DNS load-balancing management layer in front of EC2 [31].

2.2.3 HTTP Redirection

Another way to rendezvous clients with servers is to use a front-end server whose only function is HTTP redirection [11]. Microsoft's "Live Mail" (formerly Hotmail) exemplifies this strategy. Users access `mail.live.com`. If they have a login cookie, they are given an HTTP redirect to a specific host in the Live Mail farm, such as `by120w.bay120.mail.live.com`. (Users who are not logged in are redirected to a login page.) All interactions beyond the first redirection happen directly with that machine. The HTTP redirector, of course, can base its redirection decisions on instantaneous load and availability information about the servers in its farm.

This solution is attractive for two reasons. First, it introduces very little overhead: the redirector is not involved in the session other than providing the initial redirect. Second, redirection doesn't take much time; a single redirection server can supply redirections to a large number of clients very quickly. (URL forwarding services such as `tinyurl.com` and `snipurl.com` demonstrate of this: individual redirections take very little time, so they can easily provide redirection service at Internet scales.)

2.2.4 L4 or L7 Load Balancing

In both L4 and L7 load balancing, a machine answers all requests on a single IP address, and spreads the request streams out to back-end servers to balance load. The client appears to communicate with a single logical host, so there is no dependency on client behavior. Faults can be quickly mitigated because the load-balancing machine can route new requests away from a failed server as soon as the failure is known.

L4 (layer 4) load balancing is also known as "reverse network address translation (NAT)". An L4 balancer inspects only the source IP address and TCP port

of each incoming packet, forwarding each TCP stream to one of the back-end servers. L4 balancing can run at "router speeds" since the computational requirements are so modest.

L7 (layer 7) load balancing is also known as "reverse proxying." An HTTP L7 load balancer acts as a TCP endpoint, collects an entire HTTP request, parses and examines the headers, and then forwards the entire request to one of the back-end servers. L7 balancing requires deeper processing than L4, but provides the balancer the opportunity to make mapping decisions based on HTTP-level variables (such as a session cookie), or even application-specific variables.

One important disadvantage of load balancers is that high-performance load balancing switches can be very expensive (tens to hundreds of thousands of dollars), difficult to fit into a garage budget. However, there are lower-cost options. First, there is free software, such as Linux Virtual Server [22], and commodity software, such as Microsoft Internet Security and Acceleration Server [32], that implement L4 and L7 load balancing, though they are less performant than dedicated hardware. The second option is a service introduced in October of 2007 by FlexiScale [35]. They combine on-demand virtual machines with fractional (utility) access to a high-performance L4/L7 load balancing switch. To our knowledge, this is the only current offering of a load balancing switch billed as a utility.

2.2.5 Hybrid Approaches

The techniques described above can be combined to offset their various limitations.

One example above (Section 2.2.1) splits a service, such as a video sharing site, into a low-bandwidth active component managed by a load-balanced cluster, and a high-bandwidth static component served out of the SDN.

Alternatively, consider a DNS cluster of L4/L7 load balancers: Each L4/L7 cluster is fault-tolerant mitigating DNS' sluggishness to recover from back-end faults; and the entire configuration can scale beyond the limits of a single L4/L7 cluster.

3 Analysis of the Design Space

In this section, we analyze the tradeoffs a garage innovator is likely to encounter when building a scalable service, using one of the design templates we reviewed in Section 2.2, and implemented on top of the building blocks we reviewed in Section 2.1. Our analysis is drawn from both a series of micro-benchmark experiments, fleshed out in Section 4, and lessons learned from our own implementations of garage-style services that were subjected to real flash crowds, described in Section 5.

The important design criteria are:

Criterion	Design			
	Bare SDN	HTTP Redir.	L4/L7 Load Bal.	DNS Load Bal.
§3.1: Application Scope	Static HTTP	HTTP	All	All
§3.2: Scale Limitation	Very large	Client arrival rate	Total traffic rate	Unlimited
§3.3: Client affinity	N/A	Consistent	Consistent	Inconsistent
§3.4: Scale-Up Time	Immediate	VM Startup Time (about a minute)	VM Startup Time (about a minute)	VM Startup + DNS TTL (5-10 minutes)
§3.4: Scale-Down Time	Immediate	Session Length	Session Length	Days
§3.5: Front-End Node Failure: Effect on New Sessions	N/A	Total Failure	Total Failure	Major Failure
§3.5: Front-End Node Failure: Effect on Estab. Sessions	N/A	No effect	Total Failure	Rare effect
§3.5: Front-End Node Failure: Effect on New Sessions (m redundant front-ends)	Unlikely	long delay for $1/m$ th sessions?	long delay for $1/m$ th sessions?	Short delay (§4.2)
§3.5: Front-End Node Failure: Effect on Estab. Sessions (m redundant front-ends)	Unlikely	No effect	$1/m$ th sessions fail	A few sessions see short delay
§3.6: Back-End Node Failure: Effect on New Sessions	Unlikely	No effect	No effect	long delay for $1/n$ th of sessions
§3.6: Back-End Node Failure: Effect on Estab. Sessions	Unlikely	User-recoverable failure	Transient failure	long delay for $1/n$ th of sessions

Table 1: A summary of the tradeoffs involved in different scaling architectures. Section 2.2 describes the four designs. In this table, “Front-End” refers to the machine that dispatches clients to the server that will handle their request. (In the case of DNS load balancing, this refers to the DNS server.) “Back-End” refers to one of the n instances (for example, of a web server) that can handle the client’s request. A full discussion of the criteria is in Section 3.

Application scope. Does this design work only for the web, or for every kind of Internet service?

Scale limitations. What is the crucial scale-limiting factor of the design?

Client affinity. Different load distribution strategies have different effects on how consistently a client binds to a particular server. What behavior must the garage innovator expect?

Scale-up and Scale-down time. How long does it take to expand and contract the server farm?

Response to failures. How many users do typical failures affect? What’s the worst-case effect of a single failure?

Table 1 has a concise summary of the discussion in this section. Roughly speaking, the rows of Table 1 correspond to paper sections §3.1–§3.6; the columns correspond to §2.2.1–§2.2.4.

3.1 Application Scope

The first and most basic question of any scalability strategy is: *will it work with my application?*

The Bare SDN has the most restrictive application model. Services like S3 have a specific, narrow inter-

face: they serve static content via HTTP. Though many sites contain large repositories of static content, most are not *exclusively* static, so the bare SDN is rarely the complete story.

The HTTP Redirector is slightly wider in scope. Redirection only works with HTTP (and, perhaps, a very small number of other protocols with a redirection primitive). However, unlike with an SDN, clients can be redirected to servers that can run user code, facilitating dynamic web services. However, this technique does not work for protocols that have no redirection primitive, such as FTP, SSH, SMTP and IRC.

L7 load balancers understand a specific application-layer protocols such as HTTP and FTP, and thus are constrained by their vocabulary.

DNS load balancing and **L4 load balancers** work with all applications. DNS works broadly because most applications use a common resolver library that iterates through DNS A-records until it finds one that works. L4 load balancers work broadly because they operate at the IP layer, thus working with any application protocol running over IP without cooperation from the client.

3.2 Scale Limitation

A crucial consideration is scaling limits: what bottleneck will we first encounter as the load increases?

SDNs have implementation bottlenecks that are, to put it simply, not our problem. The two main SDNs today have service level agreements that make scaling *their* responsibility. A garage innovator can pretend the capacity is infinite.

HTTP redirection is involved only at the beginning of each client's session, and thus its scaling limit depends on the typical duration of client sessions. Longer sessions amortize the cost of doing the redirection during session setup. Our experience is that redirection is so cheap that, for typical applications, it scales to thousands or more clients. To evaluate this hypothesis, we built and measured a load-balancing, server-allocating HTTP redirector, described in Section 4.1.

L4/L7 load balancing is limited by the forwarder's ability to process the entire volume of client traffic. How this limit affects a web service depends on the service's ratio of bandwidth to computation. Sites that do very little computation per unit of communication, such as video sharing sites, are likely to be quickly bottlenecked at the load balancer—especially a load balancer that is built from commodity hardware. Conversely, sites that compute-intensive and communication-light, such as a search engine, will be able to use an L4 load balancer to support far more users and back-end servers.

DNS load balancing has virtually no scaling limit for garage innovators. Our experiment described in Section 4.5 suggests that thousands of back-end servers can be named in a DNS record. A service that requires more than several thousand servers is safely out of the garage regime. One tangle is that the success of DNS-based load balancing depends on sane client behavior. Most clients behave well, selecting a random server from a DNS record with multiple IP addresses. Unfortunately, as we will see in Section 4.4, some client resolvers defeat load balancing by apparently *sorting* the list of IP addresses returned, and using the first!

3.3 Client Affinity

Consider a single client that issues a long series of related requests to a web service. For example, a user might log into a web-based email service and send dozens of separate requests as he reads and writes mail. The implementation of many such applications is easier and more efficient if related requests are handled by the same server. Some load balancing techniques can enforce this property; others do not.

SDNs provide a simple contract: the client requests an object by URI, and the SDN delivers the entire object in a single transaction. The SDN is responsible for fulfilling the request, regardless of where it arrives.

HTTP redirection provides strong client affinity because a user is sent to a specific machine at the beginning of each session. The client will continue using that server until the user explicitly navigates to a new URL.

L4 balancers, in principle, could map each client by its source IP address. In practice, however, NAT may hide a large population of clients behind a single IP address, confounding the balancer's ability to spread load. Conversely, a user behind multiple proxies may send a series of requests that appear to be from different IP addresses. Absent source address mapping, the L4 balancer can provide no affinity, and thus the back-end service accepts the responsibility to bring the data to the server (Section 5.3) or vice versa (Section 5.2).

L7 balancing works transparently for any client that implements HTTP cookies or a service-specific session protocol well enough to maintain the session when interacting with a single server. Because it can identify individual sessions, the L7 balancer can enforce client affinity.

In the case of **DNS load balancing**, however, clients and proxies misbehave in interesting ways that confound client affinity. DNS resolvers seem to cluster around a particular address when we would rather they didn't (Section 4.4); and many browsers implement "DNS pinning" to mitigate cross-site scripting attacks [18]. Despite these properties, client browsers cannot be relied upon to show affinity for a particular server, as we describe in Section 4.6.

In summary, HTTP redirection and L7 balancing can enforce client affinity. For L4 balancing and DNS balancing, we recommend that the service assume the front end offers no client affinity.

3.4 Scale-Up and Scale-Down Time

Scale-up time of the server farm is a crucial concern: can a farm grow quickly enough to meet the offered load of a flash crowd? Scale-down time, on the other hand, does not usually affect the user experience; it is important only for efficiency's sake. If the system can not quickly scale back down after a period of high load, the service is needlessly expensive: our innovator is paying for resources she doesn't need.

Bare SDNs have essentially instantaneous scale-up and scale-down time. Services like S3 always have enormous capacity to handle the aggregate load of all their customers. The magic of statistical multiplexing hides our garage innovator's peak loads in the noise.

HTTP Redirectors and **L4/L7 Load Balancers** have identical scale-up and scale-down behavior. Once the decision to increase capacity has been made, these systems must first wait for a new virtual machine instance to be created. Anecdotally, our experience with Amazon EC2 has shown this usually happens in about one

minute. The moment the VM has been created, the load balancers can start directing traffic to them.

Scale-down time is a bit more difficult to pin down precisely. Once a scaling-down decision has been made, the load balancers can immediately stop directing new (incoming) sessions to the machines that are slated for termination. However, *existing* sessions, such as slow HTTP transfers or mail transactions, will likely be in progress. To avoid user-visible disruption, these long-lived sessions must be allowed to complete before shutting down the machine. In many cases, transport-level session terminations are hidden from the user by clients that transparently re-establish connections. A pedant might insist that the scale-down time is really the *worst-case* session length. Chen et al. [7] explore how to minimize the number of disrupted long-lived sessions by biasing the allocation of new sessions to servers even before the scale-down decision has been made.

DNS load balancing is the most problematic in its control over load balancing. Recall that in this scheme, back-end server selection is performed by the *client*—it selects a host from the multiplicity of DNS A-records it receives. These records are cached in many places along the path from the garage’s DNS server to the client application. Unlike the situation with HTTP redirectors and L4/L7 load balancers, the entity making the load balancing decision does not have a fresh view of the servers that are available. This has a negative effect on scale-up time. While the new DNS record can be published immediately, many clients will still continue using the cached record until after the DNS TTL expires. Fortunately, the nature of a flash crowd means that most of the traffic is new. New users are more likely to have cold caches and thus see the new servers.

Scale-down time for DNS load balancing is even more problematic. As the disheartening study by Pang et al. showed [28], nearly half of clients inappropriately cache DNS entries far beyond their TTL, sometimes for as long as a day. Anecdotal, we have seen this effect in our deployments—servers continue to receive a smattering of requests for several days after they’re removed from DNS. Therefore, to ensure no clients see failed servers, we must wait not just for the worst-case session time, but also the worst case DNS cache time.

3.5 Effects of Front-End Failure

Many distributed systems are vulnerable to major disruption if the nodes responsible for load balancing fail. We call the first-encountered node in the load balancing system the “front end.” What happens when front-end nodes fail?

The SDN, being a large-scale and well-capitalized resource, typically has multiple, redundant, hot-spare load balancers as its front end. Failure is unlikely.

L4 and L7 load balancers are highly susceptible to failure; they forward every packet of every session. If a single node provides load-balancing and fails, the system experiences total failure—all new and existing sessions stop. If there are m load balancers, the effect of a failure depends on how the front-ends are, themselves, load balanced. If they are fully redundant hot spares (common with expensive dedicated hardware), there will be no effect. Companies like FlexiScale do offer this service, at utility pricing, as we mentioned in Section 2.2.4.

More commonly, redundant L4/L7 front-ends are DNS load-balanced. In this case, $1/m$ th of sessions experience up to a three minute delay (see our experiment in Section 4.3) until they fail over to another front-end. $1/m$ is often large because m is often small; m is small because front-end redundancy is typically used for failure resilience, not scaling.

HTTP Redirectors fail in almost exactly the same way as L4/L7 load balancers, with one exception: *existing* sessions are not affected. The redirector is no longer in the critical path after a session begins. *New* sessions have the same failure characteristics as in the L4/L7 balancer case.

DNS load balancing is also highly susceptible to failure if there is only one authoritative nameserver. Name-server caches will be useful only in rare cases because the TTLs must be kept low (so as to handle scale-up, scale-down, and back-end node failures). Few new sessions are likely to succeed. Existing TCP streams will continue to work, however, since DNS is no longer involved after session setup.

This gloomy-sounding scenario for DNS can, however, be easily overcome. As we mentioned in Section 2.1.4, DNS replication services are plentiful and cheap. Widespread replication of DNS is easy to achieve, even on a garage budget. Furthermore, as we demonstrated in one of our microbenchmarks (Section 4.2), most DNS clients recover from DNS server failures *extremely* quickly—in our experiment, 100% of DNS clients tested failed over to a secondary DNS server within 2.5 seconds. This means that a front-end failure has virtually no observable effect.

This scenario is the most compelling argument for DNS load balancing: At very low cost, there is no single point of complete failure.

3.6 Effects of Back-End Failure

We next consider how the load-balancing scheme affects the nature of user-visible disruptions to the service when a back-end node fails. Recall that by “back-end” node, we mean a member of the n -sized pool of machines that can accept connections from clients. (Distinguish this from the “front end” of the load balancing scheme, which is the load balancer itself.)

The SDN is managed entirely by the service provider, so its back-end failures are not a concern to the garage innovator. The architecture of S3 is said to be highly redundant at every layer. Our experience is that occasional writes do fail (perhaps 1% of the time) but nearly always work on the first retry.

HTTP redirector and **L4/L7 load balancers** offer the best performance for garage-written services in the case of back-end node failure. *Newly arriving sessions* see no degradation at all: the redirector or load balancer knows within moments that a back-end node has failed, and immediately stops routing new requests to it. *Existing sessions* see only transient failures. Users of an HTTP-redirection service who are stuck on a dead node might need to intervene manually (that is, go back to the dispatching URL, such as `mail.live.com`). Load-balanced services potentially see only a transient failure. If the client tries to re-establish the failed TCP connection to what it thinks is the same host, it will be transparently forwarded to an operational server.

DNS load balancing suffers the worst performance in the case of back-end failure. Unlike load balancers and HTTP redirectors, which stop requests to a failed server immediately, DNS load balancing can continue to feed requests to a failed server for more than a day, as we saw in the previous section. If n servers are deployed, $1/n$ th of sessions will be unlucky enough to pick the failed server. Unfortunately, when this happens, our experiments have shown that some combinations of client and proxy take up to three minutes to give up and try a different IP address; see Section 4.3.

4 Experimental Micro-Benchmarks

In this section, we flesh out the details of some of the more complex experiments we performed in support of our analysis in the previous section.

4.1 An EC2-Integrated HTTP Redirector

To better understand HTTP redirection performance, we built and evaluated a load-balancing HTTP redirector. It monitors the load on each running service instance, re-sizes the farm in response to load, and routes new sessions probabilistically to lightly loaded servers.

Servers send periodic heartbeats with load statistics to the redirector. The redirector uses both the presence of these heartbeats and the load information they carry to evaluate the liveness of each server. Its redirections are probabilistic: the redirector is twice as likely to send a new session to one server whose run queue is half as long as another's. When the total CPU capacity available on servers with short run queues is less than 50%, the redirector launches a new server; when the total CPU capacity is more than 150%, the redirector terminates a server whose sessions are most stale.

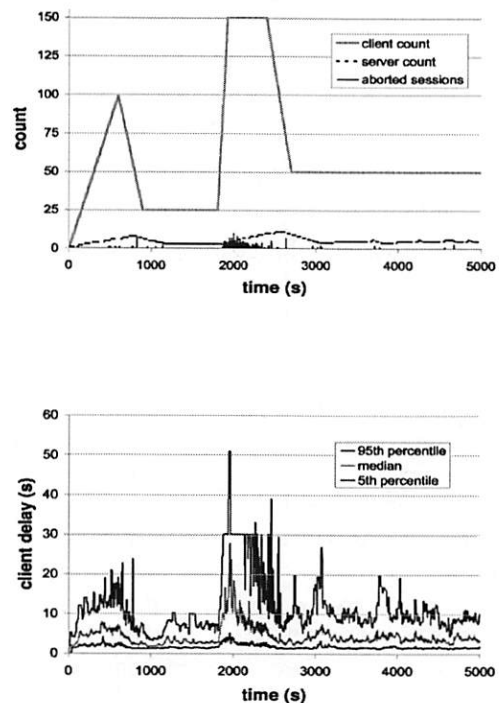


Figure 1: HTTP redirect experiment (§4.1). As client load spikes, the redirector launches new servers and directs new sessions to them.

The servers in the experiment run our Inkblot web service (see Section 5.3). The client load is presented from a separate machine. Each client simulates a user session with a state machine: It logs into the service, accesses random features for a random duration, and then logs out. Each such client session first accesses the service by the redirector's URL. Each session is recorded as having completed successfully or having been interrupted by a failure such as an HTTP timeout.

The top line in Figure 1a shows how we varied the number of simulated clients as the experiment evolved, and the dotted line shows the number of servers allocated by the redirector to handle the demand.

Figure 1b shows the 5th, 50th, and 95th percentiles of client latency; the server run queues (not shown) track these curves closely as Little's result predicts [20]. Our simplistic redirector allocates one server at a time, bounding its response rate to a slope of one server every 90–120 seconds. Around 2000 seconds, the load grows quite rapidly, and client response time suffers badly, with many sessions aborting (bottom curve in Figure 1a).

During these experiments, the redirector consumed around 2% of its CPU when serving 150 clients. Thus, for this application, we expect to be able to serve 7,500

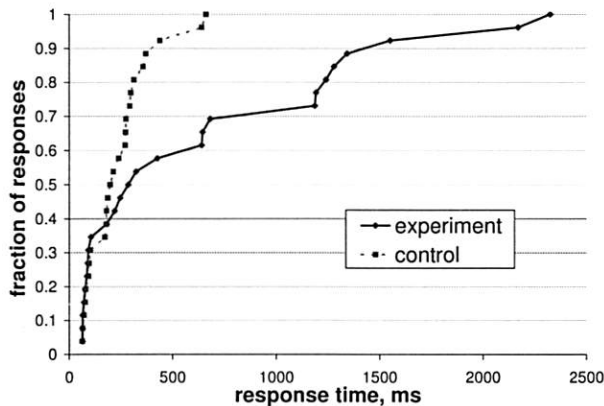


Figure 2: DNS servers fail over very quickly when an upstream server fails (§4.2).

client sessions per redirector node. However, this ratio is sensitive to our choice of client workload simulation parameters. In an application where sessions are longer and require only a single redirection during initialization, the overhead of the redirector will shrink. The important observation is that the redirector is simple, cheap, and applies to many web-based applications.

4.2 DNS server failover behavior

To determine the delay associated with nameserver failure, we configured a DNS subdomain with two NS records: one legitimate and the other pointing at an IP address with no DNS server. As a control, we repeated the same experiment with a fresh subdomain containing only the NS record for the functioning server. In each case, we sent recursive queries to 26 geographically-distributed public DNS servers at seven ISPs. Figure 2 shows the cumulative distribution of response times. The worst-case response time was less than 2.5s; of course, this cost is incurred only on a DNS cache miss.

4.3 Web client DNS failover behavior

To determine the user-visible effect of DNS-based recovery from failed back-end servers, we set up the following experiment: A browser script resolves a series of fresh hostnames, each of which resolves to two fresh IP addresses. One IP address has a live web server; the other does not. We vary the numerical order of the IPs to avoid biasing misbehaving resolvers (Section 4.4). The script repeats this experiment 20 times.

Often, the browser is lucky and tries the operational IP address first. When it doesn't, the user experiences delays from 3 to n seconds, as shown in Table 2. The delay appears to be coupled to the OS resolver library rather than the browser. We repeated the experiments with the browser behind a Squid 2.6.16 proxy running on Linux 2.6.17, where the delay seemed to be defined by the proxy.

The initial experiment used fresh names and IP addresses to ensure the client DNS resolvers always had cold cache behavior: If resolvers cache knowledge of failure, then the experiment would show no delay. Surprisingly, in some configurations (both the unproxied browsers on Linux and any browser using a Linux Squid cache) the resolver behaved *worse* with a warm cache than with a cold one. Queries that were slow in the first run showed no change. However, on queries that were fast in the first run exhibited delays of 20–190s. We are unsure why these configurations exhibit different timeout behavior for a cold request than for a warm one.

To summarize: Depending on operating system and proxy cache configuration, using DNS to failover from a non-responsive server causes clients experience delays from 3 to 190 seconds.

4.4 Badly-behaved resolvers defeat load balancing

This same experiment also revealed that in every configuration except the two browsers running uncached on Windows XP, the browser (or proxy) always tried the lower-numbered IP address before the higher-numbered one, regardless of the order they were returned by the DNS server. This behavior has important implications for using DNS as a load-balancing mechanism.

A conventional DNS server always returns the same complete list of machines. With this approach, many machines will flock to the lowest-numbered IP address in the list. Ignoring proxies, the 8% of the desktop market running MacOS X will cause significant problems once the clients present 12 server's worth (that is, $1/0.08$) of load.

Alternately, a custom DNS server might return only a single IP address to each query, exchanging fault tolerance for load balancing. A further refinement returns k servers of the available n . This provides fault-tolerance against $k - 1$ failures, but ensures that even the badly-behaved resolvers spread themselves across the first $n - k$ machines.

4.5 The maximum size of DNS replies

The scalability of DNS load balancing is potentially limited by the number of "A" (Address) records that can be put into a single DNS reply. The DNS RFC only guarantees transmission of a 512-byte reply when in UDP mode [23][§2.3.4], and some past work has reported that DNS clients do not always fall back to using TCP to retrieve larger responses [16]. At 16 bytes per A-record, and counting other overhead, a 512 byte DNS reply would limit us to about 25 back-end servers—potentially a bottleneck.

To test this limit in practice, we populated a single name with over 2200 A records, and then probed it via

OS	Browser	no-proxy delay (s)	proxied delay (s)
Windows XP	Firefox 2.0.0.6	21	9–18
Windows XP	IE 6.0	21	9–12
Linux 2.6.17	Firefox 2.0.0.3	3	9
Linux 2.6.17	Epiphany 2.16	3	<i>not measured</i>
MacOS X 10.4.10	Firefox 2.0.0.11	15–75	9–17
MacOS X 10.4.10	Safari 2.0.4	12–75	9–18

Table 2: DNS timeout test for web sites that have multiple A records. The table reports timeouts observed with various browser and OS combinations before the client attempts to use a second IP address if no reply is received from the first.

26 public DNS servers in seven ISPs. Every server correctly returned the entire list, indicating its client had used TCP DNS transfers to fetch the complete reply from our nameserver. We have not explored whether some clients (e.g., web browsers) will fail to fetch large replies. This result suggests that DNS implementations pose no limit to the number of back-end servers served by a DNS load balancing solution; indeed, other limits of DNS behavior obviate including the entire server list in DNS replies (Section 4.4).

4.6 Client affinity observations

One of our real services (Section 5.2) is equipped to tolerate client non-affinity, even though we expected affinity to be the common case. We instrumented the service to measure client affinity, and sampled for a period of two months in a 2-server configuration. About 5% of the requests received by our web service arrive at the “wrong” server. From this, we infer that 10% of clients exhibited no affinity, and half of the requests such clients generate arrived at the “right” server by chance.

5 Application Design and Flash Crowd Experiences

In this section, we report three case studies demonstrating the design, implementation, and flash-crowd response of “garage-scalable” web services. Over the past year, our research group created three web sites that experienced sudden surges in popularity, including one that was (literally) Slashdotted. For each web site, we describe our design and implementation decisions, report how they responded to a flash crowd, and extract lessons learned from the experience.

5.1 MapCruncher

In May of 2006, our research group developed MapCruncher [10], a new web authoring tool that makes it easy for non-experts to convert their own maps into AJAX-style interactive maps.

The output of this tool is a set of static content (ordinary .png, .html and .js files) that require no special

server-side support. Serving an interactive map generated by MapCruncher requires nothing more from an HTTP server than its most basic function: reading a file off disk and returning it to a client in response to an HTTP GET. All of the dynamic behavior of the application is implemented in the client browser.

5.1.1 The Web Site

To show off MapCruncher’s functionality, we created a gallery of sample maps: about 25 gigabytes of image data spread across several hundred thousand files. We put this archive on reasonably powerful web server: a 2005-vintage Dell PowerEdge 2650 with 1GB of RAM, a 2.4 GHz Intel Xeon processor, and several SCSI disks, running IIS 6.0. We did not anticipate performance problems since IIS is performant and the content we were serving was all static.

5.1.2 The Flash Crowd

After MapCruncher was released, Microsoft published a press release describing it, which was picked up by various bloggers and Internet publications. Crowds soon arrived. Nearly all visitors viewed our map gallery, which became unusably slow. Our server logs showed that, at peak, we were serving about 100 files per second. (Far more images were probably requested.)

We were surprised that our web server had failed to keep up with the request stream until we realized the dataset was many times larger than the machine’s memory cache. In addition, there was very little locality of reference. Our sample maps were enormous, and each visitor zoomed into a different, random part of one. The resulting image requests were almost entirely cache misses. We could not serve files any faster than the disk could find them. Perhaps not coincidentally, the sum of typical seek time, settling time, and rotational delay of modern disks is about 10ms (100 requests per second).

The next day, we published our maps to Amazon S3, and had no further performance problems. The lesson we learned was the power of an SDN’s statistical multiplexing: Rather than one disk seeking, the SDN spread files across huge numbers of disks, all of which can be

seeking in parallel. Rather than one buffer cache thrashing during peak load, the SDN dedicated gigabytes of buffer cache from dozens of machines to us.

S3's utility-computing cost model was as compelling to us as to a garage innovator. During a normal month, we pay virtually nothing; the nominal cost of storing our 25GB gallery is under \$4/month. In the case of a flash crowd, we pay a one-time bandwidth charge of about \$200. Statistical multiplexing makes it economically viable for Amazon to charge only for capacity used. This is much more efficient than the traditional model of paying for a fixed reserve that remains idle before the flash crowd, and yet may still be insufficiently provisioned when a crowd finally arrives.

5.2 Asirra

In April of 2007, our research group designed a web service called Asirra, a CAPTCHA that authenticates users by asking them to identify photographs as being either cats or dogs [9]. Our database has over 4 million images and grows by about 10,000 every day. Images come to us already classified (by humans) as either cat or dog, thanks to our partnership with Petfinder.com, the world's largest web site devoted to finding homes for homeless animals. Petfinder provides the Asirra project ongoing access to their database as a way to increase the number of adoptable pets seen by potential new owners.

5.2.1 The Web Site

Asirra's metadata must remain secret to maintain the security of the CAPTCHA. Consequently, Asirra is implemented as a web service rather than a distributable code library. It has a simple API that allows webmasters to integrate our CAPTCHA into their own web pages. Webmasters include Asirra's JavaScript in their HTML forms, where it first creates the visual elements of the challenge in the user's browser. It then sends AJAX requests to our web service to create a new Asirra session, retrieve one or more challenges, and submit user responses for scoring. If the response is correct, the client earns a "service ticket" (an unguessable string) and presents it to the webmaster in a special field of the HTML form. Because the client is not trusted, the webmaster's back-end then verifies that the ticket presented is valid by checking it with a different interface of Asirra's web service. The web service is implemented in Python and currently deployed at Amazon EC2.

There were several issues we considered in deciding how to enable Asirra to scale. The first was how to scalably store and serve the images themselves. The CAPTCHA's growing collection of 4 million JPEG images consumes about 100GB. Based on our experience with MapCruncher, using Amazon's S3 was an easy choice.

The second consideration was how to enable scalable access to all the metadata by every service instance. The master copy of the metadata is stored in a SQL Server database at our offices at Microsoft. However, as we discussed in Section 2.1.5, it is difficult to make a fully relational database arbitrarily scalable. We solved this in Asirra by observing that the web service treats the image metadata as read-only. (The only database writes occur off-line, when a nightly process runs to import new images into the database.) In addition, the web service does not need complex SELECT statements; when a CAPTCHA request arrives, Asirra simply picks 12 image records at random. We therefore decided to keep our fully relational (and relatively slow) SQL Server database in-house. Every time our off-line database update process runs, it also produces a reduced BerkeleyDB [27] database, keyed only by record number, that contains the relevant information about each image. (BerkeleyDB is essentially an efficient B-tree implementation of a simple key-value store.) The BerkeleyDB file is then pushed out to each running web service instance, which does local database queries when it needs to generate a CAPTCHA.

The third and most interesting design question was how to maintain session state. In between the time a user requests a CAPTCHA and the time the CAPTCHA is scored, Asirra must keep state in order to recognize if the answer was correct, and do other accounting. One possibility was to keep each session's state in its own S3 object, but we found that S3's write performance was somewhat slow; requests could take up to several seconds to complete. We next considered storing session state locally—on individual servers' disks. This led to an interesting question: how does session state storage interact with load balancing?

Client load is distributed across Asirra's server farm using the DNS load balancing technique described in Section 2.2.2. The first action performed by a client is session creation. Whichever machine is randomly selected by the client to execute this action becomes the permanent custodian of that session's state. The custodian stores the session state locally and returns a session ID to the client. The session ID has the custodian server's ID embedded in it.

As we discussed in Section 4.6, one of the disadvantages of DNS load balancing is that clients are not guaranteed to have affinity for back-end servers. Any session operation after the first one may arrive at a server other than the session's custodian. We address this by forwarding requests from their arrival server to the custodian server, if the two servers are different. That is, the arrival server finds the custodian server's ID embedded in the session ID, reissues the request it received to the custodian, and forwards the response back to the client.

Since the client is not trusted, session IDs are unguessable strings; a forged session ID will fail to find a corresponding session. Forging the identity of the custodian server will cause a request to be unnecessarily forwarded, but no corresponding session state will be found there. (As further protection, Asirra only forwards requests to servers that appear in a list of valid custodians.)

Our forwarding scheme ensures that at most two machines are ever involved in servicing a single request: the machine which receives the request from the client, and the machine that owns the session state and receives the sub-request. Asirra service is therefore readily scalable; the overhead of parallelization will never be more than 2x regardless of the total size of the farm.

In practice, we have observed lower overhead for two reasons. First, compared to satisfying a request, forwarding one takes very little time and requires no disk seeks. Even if every request required forwarding, the total overhead might not be more than 1.1x. Second, we have found that request forwarding is not the common case; as we described in Section 4.6, the rate of client affinity “failures” is about 10%.

5.2.2 The Flash Crowd

Shortly after its release, Asirra was shown publicly at an annual Microsoft Research technology showcase called TechFest. It received significant coverage in the popular press which resulted in a load surge lasting about 24 hours. During this time we served about 75,000 real challenges, plus about 30,000 requests that were part of a denial-of-service attack. Over the next few months, we saw a gradual increase in the traffic rate as sites began to use the CAPTCHA.

We learned several interesting lessons from this deployment. The first, as discussed in the previous section, was that poor client-to-server affinity was not as much of a problem for DNS-load-balanced services as we had initially feared. Second, there were some pitfalls in using EC2 as a utility for providing web services. Most problematic is that when EC2 nodes failed, as happens from time to time, they also gave up their IP address reservations. (This weakness of EC2’s service was later corrected, in April 2008.) This is a problem when using DNS load balancing. As we saw in Section 3.4, a failed node can produce user-visible service degradation until all DNS caches—even the badly behaved ones—are updated. Also, recall that local storage on EC2 nodes is fast, but not durable. Though data can be cached locally, it is vital to keep anything valuable (e.g., user data, log files, etc.) backed up elsewhere.

The denial-of-service attack provided what was, perhaps, the most interesting lesson. In the short term, before a filtering strategy could be devised, the easiest defense was simply to start more servers. The solution re-

quired no development time beyond the scalability work we’d already done, and only cost us a few extra dollars for the duration of the denial-of-service attack. Before we had a chance to implement a denial-of-service filter, the attacker became bored (and, perhaps, frustrated that his attack was no longer working) and stopped his attack. We never actually got around to implementing a denial-of-service filter—a fascinating success of “lazy development.” (The Perl community has been preaching laziness as a virtue for years!) As we will see shortly, this lesson had a surprising influence on the design of our next service.

5.3 InkblotPassword.com

In November 2007, our research group deployed InkblotPassword.com [34], a website that helps users generate and remember high-entropy passwords, using Rorschach-like images as a memory cue. The site lets users create accounts and associate passwords with inkblot images. Our site is an OpenID [30] authentication provider; users can use their inkblot passwords to log in to any web site that accepts OpenID credentials. Note that Inkblot must store dynamically generated information (the user accounts) durably. This requirement sets it apart from our previous two applications, which had static (pre-computed) databases and *ephemeral* state.

5.3.1 The Web Site

Like Asirra, we implemented Inkblot in Python. However, unlike Asirra, we spent virtually no time optimizing its performance. The denial-of-service attack we suffered taught us a valuable lesson: Now that it’s so cheap to run lots of servers for a day or two, there is no need to spend time on problems that can be solved that way. We reasoned that if our goal was simply to handle a single unexpected flash crowd, the best strategy was to forgo careful code optimization and simply plan to start plenty of extra servers for the duration of the load spike. If *on-going* popularity and high nominal load followed, careful code optimization would then be economical.

Another difference between Asirra and Inkblot was our decision to store both the persistent user database and the ephemeral session state in S3; nothing was stored on the local disk. We chose S3 for the user database, despite its slowness we observed in Asirra, because of the requirement for database persistence. Fortunately, the particular write requirements of our application permit write-behind without exposing security-sensitive race conditions, hiding most of the write delay from users. We stored ephemeral session state in S3 entirely because of our new laziness philosophy: although less efficient than using the local disk, reusing the user-state storage code led to faster development.

Like Asirra, Inkblot was implemented and tested to run on multiple servers. We deployed it with two servers, to ensure that we were exercising cross-server interaction as the common case. DNS A-records provided load balancing among the servers. Updating our institutional DNS service required interacting with a human operator, so no automatic scaling was in place.

5.3.2 The Flash Crowd

Days after its release, Network World penned an article covering Inkblot [12]. That coverage was propagated to other tech magazines and blogs.

We had the very good fortune to be in a boring meeting the next day, when one of us happened upon the article about Inkblot moments after it appeared on the front page of Slashdot. We tried clicking through the link, and found our service unresponsive. Unfortunately, this happened before we implemented the code described in Section 4.1 that automatically expands the farm in response to load. The one responsive server reported a run queue length of 137; in a healthy system, it should be below 1.

Within minutes, we spun up a dozen new servers. We submitted a high-priority DNS change request to our institutional DNS provider which was fulfilled within half an hour. The new servers saw load almost immediately after the DNS update, and the original servers recovered in another 20 minutes. (The DNS TTL was one hour at the time of the Slashdotting.) For several hours, all 14 servers' one-minute-averaged run queue lengths hovered between 0.5 and 0.9. The site remained responsive. By the end of the day, the Inkblot service had successfully registered about 10,000 new users.

We kept the extra servers up for two days (just in case of an "aftershock" such as Digg or Reddit coverage). We then removed 10 out of the 14 entries from the DNS, waited an extra day for rogue DNS caches to empty, and shut the 10 servers down. The marginal cost of handling this Slashdotting was less than \$150.

We were fortunate to survive the flash crowd so well, considering that our load-detection algorithm was "good luck." Indeed, this experience prompted us to carefully examine the alternatives for filling in the missing piece in that implementation; that examination led to the analysis and experiments that comprise this paper.

6 Conclusions

This paper surveys the contemporary state of utility computing as it applies to the low-capital garage innovator. It describes existing, utility-priced services. Our analysis characterizes four approaches to balancing load among back-end servers. We exhibit six experiments that highlight benefits and limitations of each approach. We report on our experiences deploying three innova-

tions in garage style, and how those various deployments strategies fared the flash crowds that followed.

We conclude that all four load balancing strategies are available to the garage innovator using utility resources, and that no single strategy dominates. Rather, the choice of strategy depends on the specific application and its load and fault-tolerance requirements.

7 Acknowledgements

The authors wish to thank John Douceur for reviewing drafts of this paper, and MSR's technical support organization for deployment help.

References

- [1] AKAMAI TECHNOLOGIES, INC. Edgeplatform. <http://www.akamai.com/>.
- [2] AMAZON WEB SERVICES. EC2 elastic compute cloud. <http://aws.amazon.com/ec2>.
- [3] AMAZON WEB SERVICES. SimpleDB. <http://aws.amazon.com/simpledb>.
- [4] BERRY, G., CHASE, J., COHEN, G., COX, L., AND VAHDAT, A. Toward automatic state management for replicated dynamic web services. In *Netstore Symposium* (Oct. 1999).
- [5] BRISCO, T. DNS Support for Load Balancing. RFC 1794 (Informational), Apr. 1995.
- [6] CHALLENGER, J., IYENGAR, A., WITTING, K., FERSTAT, C., AND REED, P. A publishing system for efficiently creating dynamic web content. In *INFOCOM 2000 Conference* (Mar. 2000).
- [7] CHEN, G., HE, W., LIU, J., NATH, S., RIGAS, L., XIAO, L., , AND ZHAO, F. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *to appear, Networked Systems Design & Implementation* (2008).
- [8] DATE, C. J. *An Introduction to Database Systems*, 8th ed. Addison-Wesley, 2004.
- [9] ELSON, J., DOUCEUR, J. R., HOWELL, J., AND SAUL, J. Asirra: a CAPTCHA that exploits interest-aligned manual image categorization. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS), Alexandria, Virginia, USA* (2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM, pp. 366–374.
- [10] ELSON, J., HOWELL, J., AND DOUCEUR, J. R. Mapcruncher: integrating the world's geographic information. *Operating Systems Review* 41, 2 (2007), 50–59.

- [11] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.
- [12] FONTANA, J. Forget sticky notes, microsoft using inkblots as password reminders. <http://www.networkworld.com/news/2007/120407-microsoft-inkblots-passwords.html>, Dec 2007.
- [13] GOLDSZMIDT, G., AND HUNT, G. Scaling internet services by dynamic allocation of connections. In *Proc. Integrated Management (IM 99)* (May 1999).
- [14] GRANITE CANYON GROUP, LLC. Public DNS. <http://www.granitecanyon.com/>.
- [15] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for internet service construction. In *Fourth Operating Systems Design and Implementation* (Oct. 2000).
- [16] GUDMUNDSSON, O. DNSSEC and IPv6 A6 aware server/resolver message size requirements. draft-ietf-dnsext-message-size-00, June 2000.
- [17] HENDERSON, C. *Building Scalable Web Sites*. O'Reilly Media, 2006.
- [18] JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. Protecting browsers from DNS rebinding attacks. In *Computer and Communications Security* (October 2007).
- [19] JUL, E., LEVY, H., HUTCHINSON, N., , AND BLACK, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems* 6, 1 (Feb. 1988), 109–133.
- [20] KLEINROCK, L. *Queueing Systems*, vol. I. John Wiley & Sons, Inc, 1975.
- [21] LIMELIGHT COMMUNICATIONS INC. Limelight networks. <http://www.limelightnetworks.com/>.
- [22] LINUX VIRTUAL SERVER PROJECT. <http://www.linuxvirtualserver.org/>.
- [23] MOCKAPETRIS, P. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987.
- [24] NEUSTAR. UltraDNS. <http://www.neustarultraservices.biz/solutions/externaldns.html>.
- [25] NIRVANIX INC. Nirvanix Web Services API developer's guide v1.0. <http://developer.nirvanix.com/sitefiles/1000/API.html>, Dec. 2007.
- [26] NOELDNER, C., AND CULVER, M. Scalable media hosting with Amazon S3. Amazon Web Services Developer Connection, <http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1073>, Nov. 2007.
- [27] ORACLE CORPORATION. Berkeley DB. <http://www.oracle.com/database/berkeley-db.html>.
- [28] PANG, J., AKELLA, A., SHAIKH, A., KRISHNAMURTHY, B., AND SESHAN, S. On the responsiveness of DNS-based network control. In *Internet Measurement Conference* (2004), A. Lombardo and J. F. Kurose, Eds., ACM, pp. 21–26.
- [29] RAYBURN, D. Cdn pricing data: What the cdns are actually charging for delivery. <http://tinyurl.com/25muah>.
- [30] RECORDON, D., AND REED, D. OpenID 2.0: a platform for user-centric identity management. In *Digital Identity Management* (2006), A. Juels, M. Winslett, and A. Goto, Eds., ACM, pp. 11–16.
- [31] RIGHT SCALE LLC. Rightscale dashboard. <http://info.rightscale.com/>.
- [32] SCHINDER, T. *ISA Server 2006 Migration Guide*. Elsevier, 2007.
- [33] SCHLOSSNAGLE, T. *Scalable Internet Architectures*. Sams Publishing, 2006.
- [34] STUBBLEFIELD, A., AND SIMON, D. Inkblot authentication. Technical report MSR-TR-2004-85, Microsoft Research, Aug. 2004.
- [35] XCALIBRE COMMUNICATIONS LTD. Flexiscale. <http://www.flexiscale.com/>.
- [36] YU, H., AND VAHDAT, A. Design and evaluation of a continuous consistency model for replicated services. In *International Conference on Distributed Computing Systems (ICDCS)* (Apr. 2001).

Remote Profiling of Resource Constraints of Web Servers Using Mini-Flash Crowds

Pratap Ramamurthy
UW-Madison

Vyas Sekar
CMU

Aditya Akella
UW-Madison

Balachander Krishnamurthy
AT&T-Labs Research

Anees Shaikh
IBM Research

Abstract

Unexpected surges in Web request traffic can exercise server-side resources (e.g., access bandwidth, processing, storage etc.) in undesirable ways. Administrators today do not have requisite tools to understand the impact of such “flash crowds” on their servers. Most Web servers either rely on over-provisioning and admission control, or use potentially expensive solutions like CDNs, to ensure high availability in the face of flash crowds. A more fine-grained understanding of the performance of individual server resources under emulated but realistic and controlled flash crowd-like conditions can aid administrators to make more efficient resource management decisions. In this paper, we present mini-flash crowds (MFC) – a light-weight profiling service that reveals resource bottlenecks in a Web server infrastructure. MFC uses a set of controlled probes where an increasing number of distributed clients make synchronized requests that exercise specific resources or portions of a remote Web server. We carried out controlled lab-based tests and experiments in collaboration with operators of production servers. We show that our approach can faithfully track the impact of request loads on different server resources and provide useful insights to server operators on the constraints of different components of their infrastructure. We also present results from a measurement study of the provisioning of several hundred popular Web servers, a few hundred Web servers of startup companies, and about hundred phishing servers.

1 Introduction

As Web-based applications on the Internet grow in popularity, their providers face the key challenge of determining how to provision server-side resources to provide consistently good response time to users. Ideally, these resources, such as processing and memory capacity, database and storage, and access bandwidth, should be provisioned to deliver satisfactory performance under a broad range of operating conditions. Since an operator’s ability to predict the volume and mix of requests is

often limited, this can be difficult. Hence, large providers who can afford it typically resort to over-provisioning, or employ techniques such as distributed content delivery or dynamic server provisioning, to minimize the impact of unexpected surges in request traffic. Smaller application providers may trade-off robustness to large variations in workload for a less expensive infrastructure that is provisioned for the expected common case.

This approach still leaves operators without a sense of how their application infrastructure will handle large increases in traffic, due to planned events such as annual sales or Web casts, or unexpected flash crowds. While these events may not occur frequently, the inability of the infrastructure to maintain reasonably good service, or at least degrade gracefully, can lead to significant loss of revenue and dissatisfied users. Without comprehensive stress testing that would likely disrupt service, there is no way for providers today to observe the performance of their sites under heavy load in a controlled way to inform their preparation for unexpected traffic increases.

In this paper, we present the design, implementation and evaluation of a new profiling service that helps operators better understand the ability of their Internet applications to withstand increased request load. Our mini-flash crowd (MFC) mechanism sheds light on bottlenecks in the application infrastructure by quantifying the number and type of simultaneous requests that affect response time by taxing different parts of the server set-up. Using the service, an application provider can compare the impact of an increase in database-intensive requests versus an increase in bandwidth-intensive requests. The operator could then make better decisions in prioritizing additional provisioning, or take other actions (e.g., introduce request shaping).

The MFC technique is based on a phased set of simple, controlled probes in which an increasing number of clients distributed across the wide-area Internet make synchronized requests to a remote application server. These requests attempt to exercise a particular part of the infrastructure such as network access sub-system, storage sub-system, or back-end data processing subsystem. As the number of synchronized clients increases, one or

more of these resources may become stressed, leading to a small, but discernible and persistent, rise in the response time. Inferences can now be made about the relative provisioning of the resources. The number of clients making simultaneous requests is increased only up to a set maximum – if no change in the response time is observed, we label the application infrastructure as unconstrained. Such a conservative approach allows MFC to reveal resource constraints while limiting its intrusiveness on the tested sites.

The MFC technique can be thought of as a “black-box” approach for determining the resource limitations of a Web server, or for uncovering performance glitches, vulnerabilities, and configuration errors. The salient features of the approach are: (i) light-weight requests that have minimal impact on, and involvement from, production servers; (ii) use of real, distributed clients that test the deployed application infrastructure while accurately reflecting client access bandwidth and the effects of wide-area network conditions; and (iii) ability to work with a broad range of Web applications with little or no modification, while providing some tunability to run more application-specific tests.

We validate the effectiveness of MFC in tracking server response times using synthetically generated response time curves (i.e., as a function of the request load). We also study the ability of MFC to exercise specific server resources by running experiments against a real Web application in a controlled lab setting.

Beyond the MFC technique itself, a principal contribution of the paper is an application of the MFC service to a top-50 ranked commercial site and three university sites with the active cooperation of the site operators. The operators gave us the server logs for the experiments – we ascertained that MFC requests were well-synchronized and studied the impact of background traffic on MFC. The operators confirmed MFC’s non-intrusive nature and found it very useful in uncovering new (or confirming suspected) issues with their infrastructure.

The lab-based experiments and experiments on co-operating sites demonstrate the usefulness of applying MFCs to production Web applications. They also show that the granularity of information that black-box testing can reveal is limited. MFCs are able to isolate resource constraints at a “sub-system” level, such as the storage subsystem and database access subsystem, which includes both hardware and software components of the sub-systems. However, providing finer-grained information to pinpoint if the constraint is due to a hardware limitation or software misconfiguration within the sub-system is difficult. This may require operator input and some site-specific changes to the MFC approach.

We also applied MFC to characterize a large number of production Web sites, classified according to their as-

signed ranking by a popular Web rating service [19]. Our empirical results show a high degree of correlation between a site’s popularity and its ability to handle a surge of either static, or database-intensive requests. Bandwidth provisioning is less well-correlated, however, with many less-popular sites having better provisioned access bandwidth than might be expected. Finally, we also present a preliminary study of the application of MFCs to other special classes of Web sites, including startup companies and sites belonging to phishers.

Although our initial application of MFCs focuses on determining how unexpected request surges affect perceived client performance, the approach can be useful in a number of other scenarios. MFCs could be used to perform comparative evaluations of alternate application deployment configurations, e.g., using different hosting providers. By tuning the request arrival pattern of clients, MFCs can be used to evaluate the impact of different request shaping mechanisms.

Section 2 describes the MFC design and implementation, and several practical issues. In Section 3, we discuss our validation study. We report on our experience running MFC with cooperating commercial and academic sites in Section 4. We describe the results of a large-scale study of production sites in Section 5. In Section 6, we discuss some extensions to MFC. We discuss related work in Section 7 and conclude in Section 8.

2 Mini-Flash Crowds

In this section, we describe the design and implementation of the Mini-Flash Crowd (MFC) approach. First, we discuss the key design requirements and the challenges in meeting these requirements. We then present an overview of MFC’s operation. Finally, we discuss implementation details and key practical issues.

2.1 Solution Requirements and Challenges

Our goal is to develop a mechanism that gives application providers useful information about the limitations of their server resources. The foremost requirement from such a mechanism is that it should accurately reflect the application’s performance under realistic load conditions; i.e., the information should be *representative* of a real flash-crowd like situation. While laboratory load testing is no doubt useful, it is difficult to re-create all of the dependencies of an Internet-facing live deployment in the lab. Traditional benchmarking approaches or load generation tools (e.g., [24]) used to test Web applications in controlled LAN settings cannot reveal effects of wide-area conditions or characteristics of the actual Internet connectivity of the clients (e.g., speed, location, diversity

etc.). More importantly, such laboratory-based experiments cannot help operators assess the impact of access infrastructure - e.g., how the bandwidth limits the performance of clients under overload, and how bandwidth-imposed constraints compare against constraints on other server-side resources.

Second, the mechanism must be *tunable* to tailor the characterization to specific operational goals. Some applications (e.g., software binary distribution) may be tolerant to large increases in response time. Others may be more sensitive to small increases in response time; for instance, it may be important to know that a 10% increase in volume of search queries caused a search engine's response time to increase by 250ms.

Finally, the approach must be *automatic*. It must require minimal (if any) input from operators about the specifics of the application and infrastructure. This requirement is crucial to deploy MFC as a *generic network service* that any content provider can sign up for and use. This requirement can be relaxed when server operators cooperate in running experiments.

The above requirements raise two challenges. First, it is challenging to develop a generic request workload that can remotely exercise specific resources or sub-systems on production server infrastructures. Second, it is difficult to exercise tight control on the load imposed on the application infrastructure so as to not cause an undesirable impact on the regular request workload at the server. Such tight control is particularly difficult to achieve when using a set of distributed clients whose requests may be affected unpredictably by the wide-area.

We use two simple insights to address these challenges. First, to exercise specific server resources, we issue concurrent requests for a particular type of content. For example, to exercise a server's network connection, we can make concurrent requests for "large" objects hosted at the server (e.g., binary executables, movie files). As we show later, this simple approach can help us isolate the impact to the granularity of server sub-systems, which include both hardware and software components of the sub-systems. To minimize the need for server-side input and to ensure generality of the approach, we crawl the content hosted on the server and automatically classify it into different content categories, using heuristics such as file name extensions and file sizes. Second, to achieve tight control over the load on the server sub-systems, we schedule client requests in a centrally coordinated manner using measurements of the network delay between each client and the target server.

2.2 Overview of the MFC Methodology

The MFC setup has a single *coordinator* orchestrating an MFC on a *target server* (Figure 1). At the coordi-

nator's command, a specified number of *participating clients* send synchronized requests to the target server. The clients log the response times for their requests and send this information to the coordinator. The coordinator uses the feedback from clients to determine how to run the MFC, to infer resource constraints, and to stop the MFC. The MFC experiment consists of an optional profiling step followed by several probing or measurement phases.

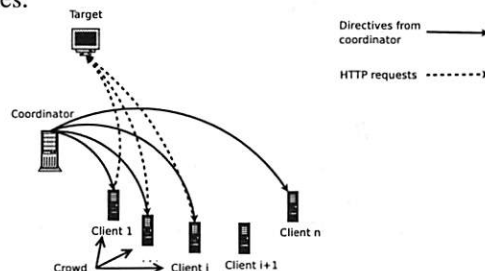


Figure 1: Structure of a Mini-Flash Crowd experiment.

2.2.1 Profiling target content

The profiling stage precedes an MFC run against a non-cooperating server, and is optional for cooperating servers. In this stage, the coordinator crawls the target site and classifies the objects discovered into a number of classes based on content-type, for example, regular/text (txt, HTML files), binaries (e.g., .pdf, .exe, .tar.gz files), images (e.g., .gif, .jpg files), and queries.

The discovered objects are also classified into two categories based on the reported object sizes (obtained by issuing a HEAD request for files and GET request for queries): *Large Objects* and *Small Queries*. These categories are selected for their expected impact on specific server resources or sub-systems (more details below).

The Large Objects group contains regular files, binaries, and images greater than 100KB in size. We identify URLs that appear to generate dynamic responses (queries) and select queries that generate a response under 15KB. These constitute the Small Queries group. Note that a small query is not necessarily a "form", but an URL with a "?" indicating a CGI script.

2.2.2 MFC Stages

After the profiling step is completed, the coordinator runs the MFC experiment in *stages*. In each stage, the MFC makes a varying number of synchronized requests for objects from a specific request category.

In the *Large Object stage*, clients request the *same* large object simultaneously, primarily exercising the server's network access bandwidth. We use a fairly large lower bound (100KB) on the size of the Large Object to allow TCP to exit slow start and fully utilize the available

network bandwidth. Since we request the same object, the likely caching of the object reduces the chance that the server's storage sub-system is exercised.

In the *Small Query stage*, each client makes a request for a unique dynamically generated object if available; else all clients request the same dynamic object. Since such queries often require interactions and computations using a back-end database, we expect that this stage will affect the back-end data processing sub-system and possibly the server CPU. We use a small upper bound (15KB) on the response size so that the network bandwidth remains under-utilized.

Finally, in the *Base stage*, clients make a HEAD request for the base page hosted on the target servers (e.g., `index.html` or `index.php`). This provides an estimate of basic HTTP request processing time at the server.

2.2.3 Epochs

Each stage of a MFC experiment consists of several *Epochs*. In epoch k , the coordinator directs N_k participating clients to issue concurrent requests of a given category to the target. Clients participating in epoch k constitute a *crowd*. The coordinator determines the particular object $O_{i,k}$ that client i should request in an epoch k . The coordinator runs each MFC stage for a preset maximum number of epochs k_{max} .

Before Epoch 1, each participating client i measures the *base response time* for downloading the objects $O_{i,1}, \dots, O_{i,k_{max}}$ from the target. Clients make these measurements sequentially so that they do not impact each other's base response time estimates.

At the end of each epoch, each client reports the normalized response time (*observed response time for the request* – *base response time for the same request*) to the coordinator. Successive epochs are separated by ~ 10 s. Based on the clients' response times for requests in epochs $1..i$, the coordinator either terminates the stage, or moves to epoch $i + 1$. The coordinator uses the following simple algorithm to decide the next step:

1. Check: For the Base and the Small Query stages, if the *median* normalized response time reported by the clients in epoch i is greater than a threshold θ , the MFC enters a "check" phase. We use the median to counter the impact of noise on the response time measurements. If the median normalized response time is X ms, this implies that at least 50% of the clients observed an X ms increase in their response time for the request.

The goal of the check phase is to ascertain that the observed degradation in response time is in fact due to overload on a particular server sub-system, and not due to stochastic effects. To verify this, coordinator creates three additional epochs, one numbered " $-$ " with $N_i - 1$ clients, and the other numbered " $+$ " with $N_i + 1$ clients,

and a third epoch which is a repeat with N_i clients. As soon as the median normalized response time in one of these additional epochs exceeds θ , the coordinator **terminates** the MFC experiment, concluding that a limitation has been found within the sub-system. If there is no visible degradation in any of the additional epochs, the check fails and the MFC **progresses** to epoch $i + 1$.

Recall that the Large Object stage is designed to exercise the server's outgoing access link bandwidth. However, depending on where the MFC clients are located relative to the target, the paths between the target and many of the MFC clients may have bottleneck links which lie several network hops away from the target server. In such cases, the median increase in response time may reflect an increase in the load on the shared network bottlenecks, and not necessarily on the server's outbound access link. To counter this, we require that a larger fraction of the clients (specifically, 90% of them) observe $> \theta$ increase in the response time in the Large Object stage.

In general, for any MFC stage, we can infer sub-system resource provisioning more accurately by relying on the 90th percentile as described above. However, it has a downside relative to using the median: we may now have to load the server's resources a bit longer before drawing an inference. As a trade-off between the requirement to be unobtrusive and the goal to accurately infer the constraints, we use the 90th percentile only for the Large Object stage, and use the median for other stages.

2. Progress: If there is no perceptible increase in the target's response time, or the check phase fails, the coordinator progresses to the next epoch where a larger number of clients participate. To ensure that the target does not face sudden load surges, the coordinator increases the size of the crowd by a small value (we choose this to be 5 or 10 in our experiments).

3. Terminate: If the check phase succeeds, or the number of participating clients exceeds a threshold, the coordinator terminates the experiment. In the latter case, the coordinator concludes that no limitations could be inferred for the particular resource or sub-system.

2.2.4 Synchronization

In a given epoch, the load on the target server is proportional to the number of concurrent requests it is serving, which directly determines the server's response time. An important requirement is that when k clients participate in an epoch, the number of concurrent MFC requests at the server is $\approx k$. One can imagine implementing a distributed synchronization protocol among the clients that can guarantee this property, but this introduces a lot of complexity. Instead, we rely on simple techniques that achieve reasonable synchronization by leveraging

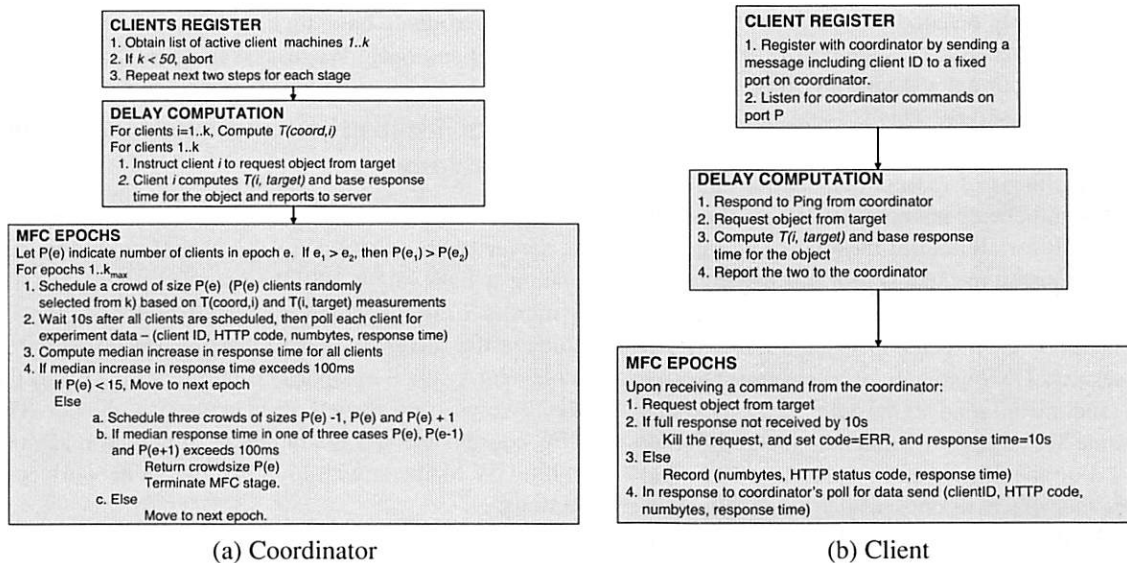


Figure 2: Figure outlining our implementation of the MFC approach.

the centralized coordinator to schedule client requests.

To ensure synchronization, the coordinator issues a command to the clients at the beginning of the experiment to measure the round-trip latency to the target server. Client i then reports the round-trip delay T_i^{target} to the coordinator. The coordinator also measures its round-trip delay to each client T_i^{coord} . Using these measurements, the coordinator schedules client requests so that they arrive at the server at roughly the same time T . Note that the actual HTTP request arrives at the server roughly at the same time as the *completion* of the 3-way SYN hand-shake. To synchronize client request arrivals, the coordinator issues a command to client i at time $T - 0.5 * T_i^{coord} - 1.5 * T_i^{target}$. Assuming that the latency between the coordinator and the clients has not changed since the initial latency estimate, client i will receive this command at time $T - 1.5 * T_i^{target}$; at this time, client i issues the request specified in the command by initiating a TCP hand-shake with the server. Again, assuming client-target latency does not change, the first byte of client i 's HTTP request will arrive at the target at time T . Since an MFC experiment spans only a few minutes, we believe that assuming that network latencies are stationary over this time-span is reasonable [26].

2.3 Implementation Specifics

We have implemented the MFC approach as described above. We use hosts from PlanetLab as the MFC clients. Our coordinator runs on an off-the-shelf Linux host located at UW-Madison. Figures 2(a) and (b) provide additional details of the functioning of clients and the coordinator. While these largely reflect the discussion in Section 2.2, a few points are worth noting.

Before conducting an MFC experiment, the coordinator checks if at least 50 distinct clients are available to run the experiment (see Figure 2(a)). The coordinator does this by verifying if at least 50 clients respond sufficiently quickly (within 1s) to a probe message. If not, the experiment is aborted. This check is important because with a small number of participating clients, we cannot claim that the MFC captures realistic wide-area conditions faced by generic Web clients accessing the target server. Ideally, we should also factor in the geographic locations of the clients, i.e., ensure that the active clients are well spread out. Our current implementation does not enforce this requirement.

Note that in order for the median and the 90th percentile response time measurements in an epoch to be statistically significant and robust to noise, we need a sufficient number of clients to be participating in the epoch in the first place. We choose this number to be 15. Thus, for all initial epochs where fewer than 15 clients participate in the measurement, the coordinator automatically progresses to the next epoch irrespective of the degradation in the response time observed in the current epoch.

Note also that the participating clients within each epoch are chosen at random (see Figure 2(a)). This is important to ensure that an observed increase in the median response time is purely due to an increase in the number of participating clients at the server, and not due to the local conditions experienced by the clients themselves (e.g., transient congestion or due to load on a client).

The client-side functionality is simple (Figure 2(b)). The client listens for commands from the coordinator and fires off HTTP requests as soon as a command is received. Clients timeout 10s after issuing an each HTTP request (see Figure 2(b)). Thus, if the target takes more

than 10s to completely respond to a request, the client kills the request and records a response time of 10s. The client computes the normalized response time as before and sends this to the server. This is to ensure that each epoch spans a bounded amount of time.

Since the timeliness of the communication between the coordinator and clients is important for synchronization, we use UDP for all control messages. We did not implement a retransmit mechanism for lost messages.

Practical Issues: In our current implementation, all the requests in the MFC are directed to a single server IP address. If a server's DNS name maps to multiple IPs, we pick one at random and send all the MFC requests to this single IP. Some Web sites may direct clients to different replicas based on the clients' geographic locations, and in such cases the MFC will only be able to identify scaling bottlenecks on the server assigned to the specific IP address chosen, and not of the Web site as a whole.

Server-side caching could also impact the observations we draw from an MFC experiment. Many servers cache Web objects and other clients not part of the MFC may request the same object concurrently. Thus, even if each MFC client requests a unique object (e.g., in the Small Query stage), we cannot guarantee that there are no caching effects. Thus we cannot ensure that the load on a specific server resource or sub-system will grow monotonically as a function of the number of requests, and that, eventually, we will observe a perceptible increase in the response time.

MFCs may require additional information to actually confirm that requests are exercising a single specific resource on a server – this is a fundamental limitation of any mechanism that relies on remote inferences. The MFC approach as such does require any data collection at the server. However, server-side support in instrumenting servers to track resource usage using utilities (such as atop or sysstat) can offer better insights.

Background traffic at the target can also impact MFC inferences. Thus, with non-cooperating sites, we suggest running MFCs at off-peak hours. With cooperating sites, site operators can indicate whether they wish to observe the limitations of the “raw” production infrastructure (under low background traffic) or the ability to handle load surges under regular operating conditions.

3 Validation Experiments

Next, we address the following questions through experiments in controlled laboratory settings: (1) Are requests from MFC clients adequately synchronous? (2) How well can MFC track the target's response behavior? (3) How effective are MFC requests at exercising intended resources at the target?

Our experiments have highlighted a few key limitations of our approach. We discuss these in Section 3.3.

3.1 Synchronization and Response Time Tracking

To answer the first two questions, we set up a simple server (with no real content and background traffic) running a lightweight HTTP server [3] on a 3.2 GHz Pentium-4 Linux machine with 1GB of RAM. We instrument the server to track request arrival times and to implement synthetic response time models. To run the MFC, we used 65 PlanetLab machines as clients. The MFC coordinator and the target are high-end machines within UW-Madison with high-bandwidth network connections.

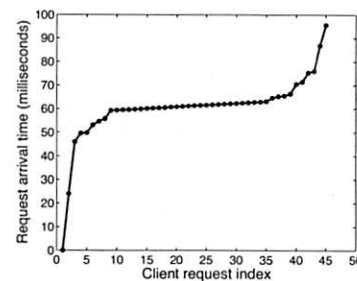


Figure 3: Arrival times at target for MFC with 45 clients
Synchronization: We logged the arrival times of each incoming HTTP request at the target server. Figure 3 shows the arrival time of each request with a crowd size of 45 clients. In these experiments, the coordinator commands the clients to make a HTTP request 15s after taking the latency measurements. About 70% of the requests arrive within 5ms of each other (clients 7 through 40), and 90% of the requests arrive within 30ms of each other (clients 3 through 43), indicating that our synchronization algorithm works quite well.

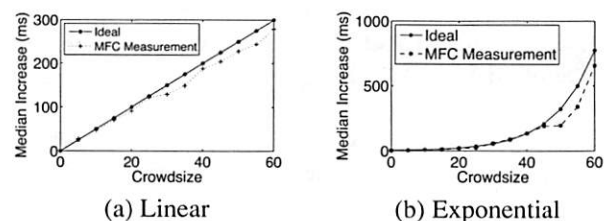


Figure 4: Tracking synthetic response time functions
Tracking Server Response Time: To validate MFC's ability to accurately track different server response behaviors, we incorporated synthetic response time models into the validation server. Each model defines the average increase in response time (relative to the base response time) per incoming request as a function of the number of simultaneous requests at the server. The response times were strictly non-decreasing functions of

the pending request queue size. We show the median normalized response times estimated by the clients for two models: linear (Figure 4(a)) and exponential (Figure 4(b)) (results with other models not shown here were similar). In both cases, the median increase in response time across the clients faithfully tracks the server's actual response time function. This shows that MFC can accurately reflect the impact observed by a remote server under controlled load surges.

3.2 Understanding Resource Constraints

Next, we examine the effectiveness of MFC in exercising specific resources at the target server. We set up a Apache 2.2 Web server (with the worker multi-processing module) on a 3 GHz Pentium-4 machine with 1GB RAM. We emulate a MFC on this target server with clients located on the same LAN as the server. For each experiment we measure the response times seen by the MFC clients and server-side resource utilization using `atop` to monitor the CPU, resident memory, disk access, and network usage. We use a maximum of 50 clients.

Large objects. In the large object workload, each MFC client requests the same 100KB object from the server. Figure 5 shows a significant increase in the median response time observed by the clients due to the network load on the server. CPU, memory, and disk utilization remain negligible during the experiment. Thus, in this case the network bandwidth constraint is primarily responsible for the increase in response time.

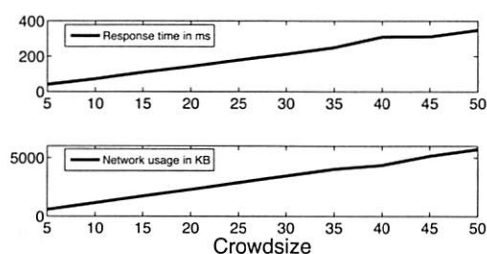


Figure 5: Same 100KB large object

Small Query workload. For emulating a dynamic object or database query workload we set up a back end database (details regarding the specific content hosted on the database are not relevant for validation). During each epoch, participating clients make the same query (thus, the responses may be cached). Each query causes the server to retrieve the same 50000 entries from a database table and return their mean and standard deviation. The query workload is not network intensive as the responses are each less than 100B. The back end database is a MySQL server with the query cache size set to 16MB.

We experimented with two server-side software interfaces for the DB back-end: the `FastCGI` [8] module

and `Mongrel` [12], a lightweight module explicitly designed for handling dynamic objects. When using `Mongrel`, we noticed that the response time stays within 10ms for crowd sizes up to 50 (not shown); the CPU utilization and memory usage stayed constant and low. However, an inefficiency in the `FastCGI` implementation¹ caused memory usage on the server to increase dramatically with the crowd size (Figure 6). Consequently, client response time also increased significantly.

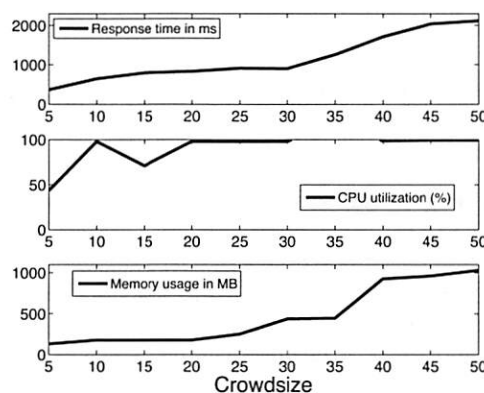


Figure 6: Small Query workload (FCGI)

3.3 Implications

The lab-based tests confirm the potential usefulness of the MFC approach in identifying resource constraints in Web servers. To put the applicability of MFC in perspective, we discuss two important factors that can affect MFC inferences: (i) impact of serial vs. parallel accessed resources, and (ii) granularity of the inferences.

An increase in the observed response time in the MFC experiment under a particular type of request can be attributed to two possibilities. One is an increase in the load on a specific server resource, where each additional request consumes a proportional fraction of the resource. The other is an increase due to server-side scheduling and resource serialization constraints, where additional requests do not impose any additional load on a resource, but create larger-sized queues of requests waiting for the resource (e.g., serialized access to a single disk). Such serialization bottlenecks can impact our ability to detect resource constraints.

Server throughput is determined by a number of factors, including hardware performance, software throughput, and server-side components used for handling requests. Our experiments show that while we may be able to isolate resource constraints at a “sub-system” granularity (e.g., disk subsystem, database subsystem, network etc), providing finer-grained information to precisely pinpoint if the constraint is a hardware or software inefficiency is difficult, especially without operator input.

MFCs are not well-suited for inferring constraints on some types of resources, such as memory buffers, which cause a sharp degradation in response time only when they are exhausted. Reaching this tipping point typically requires a very large number of simultaneous requests which conflicts our design goal of limiting intrusiveness. In our current MFC implementation, we sought to indicate coarse-grained resource constraints as a guideline for better server provisioning. Our validation results demonstrate the promise of the MFC approach to achieve this goal. Ultimately, any fine-grained analysis of resource constraints can be best understood when MFC is run in cooperation with administrators managing the target server.

4 Running MFC on Cooperating Web sites

In addition to lab-based experiments, we ran MFCs on a few cooperating commercial and academic Web sites. The server operators provided us with access logs and invaluable feedback on the accuracy and usefulness of MFC inferences. The logs enabled us to check if the requests were adequately synchronized, and examine the volume of *non-MFC* traffic (background traffic) received by the Web server during the course of our experiments. Based on this, we can see how MFC is impacted by the presence of various levels of background traffic volume.

4.1 Commercial Site

We worked with operators of a top-50 ranked commercial Web site (according to Quantcast [19]) to cooperate in our MFC experiments. The site operates a large database serving queries for users primarily in North America, and serves over a billion requests yearly. The operators allowed us to test two system configurations. One is a non-production server hosting identical content as the production server but handling minimal traffic. We call this server the QTNP (Quantcast Top-50 Non-Production) server. The administrators provided logs of all Web requests during the testing period. In addition, they also allowed us to test their production system (which we call QTP). Hence, we were able to see some details of how our experiments reflected on their site. In particular, we examine the temporal distribution of the request arrivals during our experiment (with millisecond granularity).

MFC on QTNP: We ran multiple experiments on the QTNP system, a subset of which are summarized in Table 1. We ran the three stages of the MFC on September 11 and again on September 12, 2007 (first two rows). We used a 100ms threshold for both sets of experiments.

The outcome of the experiment is similar across both the runs. For the Base stage, we observed a 100ms degra-

Expt details	Base		Small Qry		Large Obj	
	Time	Crowdsize	Time	Crowdsize	Time	Crowdsize
MFC 100ms	09/11/07	25	09/11/07	55	09/11/07	NoStop (55)
MFC 100ms	09/12/07	20	09/11/07	45	09/11/07	NoStop (55)
MFC-mr 250ms	09/21/07	40	09/21/07	90	09/21/07	NoStop (150)

Table 1: Results for QTNP non-production server. MFC traffic contributed to > 70% of all traffic at QTNP.

dation in response time when using 20-25 clients; for Small Queries, the response time crossed the threshold for a crowdsize of 45-55 clients. The Large Object stage did not impact the response time in either run (in both cases a maximum of 55 requests were issued).

We ran a slightly modified MFC, *MFC-multiple request* (MFC-mr), on QTNP on September 21, 2007 (results shown in the third row of in Table 1). In MFC-mr, each participating client opens two TCP connections to the target and sends the same request on both connections simultaneously doubling the number of MFC requests arriving at the target server.

For these experiments, we also increased the threshold to 250ms based on the QTNP operators' view that their systems would not be negatively impacted by an MFC with a higher threshold.

We had two goals in running the MFC-mr experiments on QTNP: (1) to understand the system's response when we send a larger number of simultaneous requests – particularly for the Large Object stage which showed no visible degradation under the standard MFC; and (2) to contrast the response behavior in the Base and the Small Query stages for the 100ms threshold with a higher 250ms threshold.

For Large Object, there was again no visible degradation in response time even when 150 simultaneous requests were made—the response time degraded by only a few milliseconds. This suggests that the access link is well-provisioned and the operators confirmed this.

For the Base and the Small query stages, the QTNP showed a 250ms degradation in response time with a crowd size of 40 and 90, respectively. The operators noted that the Small Query we tested involves processing on multiple servers (in addition to the back end database), and one of the servers was a known contention point. Although the degradation in the Small Query experiment confirmed a known issue, the results demonstrate that MFCs can be used to help identify and diagnose resource constraints. The Base stage response time degradation with only 40 simultaneous requests was surprising to the operators.

Finally, we examined the time synchronization of MFC-mr requests arriving at the site. We found that most of the requests in each epoch arrived closely together, within at most one second of each other. In a few of

Base			Small Qry			Large Obj		
Num reqs scheduled	Num reqs recd	Spread for 90% of reqs	Num reqs scheduled	Num reqs recd	Spread for 90% of reqs	Num reqs scheduled	Num reqs recd	Spread for 90% of reqs
25	25	0.18	25	25	0.16	25	23	1.41
40	40	1.05	40	40	1.58	40	33	0.48
55	55	0.23	55	55	0.42	55	50	0.52
75	74	0.77	75	75	0.23	75	71	1.67
100	100	0.27	100	100	0.15	100	92	0.96
125	121	0.22	125	125	0.22	125	122	1.74
175	175	0.26	175	171	0.27	175	172	2.09
225	225	0.30	225	206	0.29	225	213	1.28
275	275	0.32	275	270	0.17	275	275	1.28
325	324	0.32	325	318	0.15	325	324	2.05
375	374	0.34	375	353	0.16	375	344	3.28

Table 2: Time spread (in secs) of MFC-mr requests to QTP in the October 3 experiment. The first column shows the number of requests scheduled by the coordinator, the second column shows how many requests appeared in the server logs, and the third column shows the difference in timestamps for the middle 90% of all requests in the epoch.

the epochs, a small fraction of the requests ($< 10\%$) arrived 2-3 seconds before or after the rest. We omit the detailed results here, however we discuss the efficacy of our synchronization in greater detail in the context of the production QTP system below.

Overall, the site operators felt that MFC was a valuable tool to both analyze their local configurations and to better understand resource limitations. Although some of the results were known to the operators, MFC experiments were able to confirm them, and also uncover possible new constraints.

MFC on QTP: We ran two experiments on the production QTP system on September 27 and October 3, 2007 (results not shown for brevity) using MFC-mr, with one additional modification to the latter experiment. Each client in the first experiment made two requests in parallel as with QTNP (85 client nodes were available), while in the second experiment, each client made 5 requests in parallel (only 75 client nodes were available).

QTP received approximately 3 million and 1.6 million non-MFC requests respectively during our first and second experiments. All MFC requests were directed to a specific data center which houses 16 multiprocessor servers in a load-balanced configuration serving the requests directed to the single server IP address we used. The server logs were collected from all 16 servers.

We found that the response times in the different stages were not impacted by the MFC, even with MFC-mr with 5 requests. In fact, we did not observe even a 10ms increase in the median response time. This confirmed that the system is well-provisioned, with multiple high-end servers working in parallel. We knew from our interactions with operators that the bandwidth was well-provisioned also.

In Table 2 we examine the synchronization of MFC-

mr requests to QTP for the October 3 experiment. For the Base and the Small Query stages, the synchronization works well. For instance, in the last epoch of the Small Query stage, 90% of the 353 requests (≈ 317 requests) arrived at the server within 0.16s of each other. The synchronization was not as tight for the Large Object stage, but still reasonable with about 310 of the 344 requests in the last epoch arriving within a 3.28s time-span; 258 requests (75% of requests) arrived within 800ms.

4.2 University Sites

We ran MFC measurements on a research group Web server at a European University (labeled Univ-1) and the primary Web servers of the computer science departments of two US universities (labeled Univ-2 and Univ-3). We obtained server logs of requests arriving during the experiment time frame.

Univ-1: We ran the standard version of MFC with a 100ms threshold against the Univ-1 Web server on Aug 11, 2007. The experiment ran over a 35 minute period generating 339 out of the total of 661 (51%) HTTP requests received by the Web server. During our experiments the server had a low background traffic level of about 0.15 requests/sec. For all three stages of the MFC, we noticed that the server's response time degraded by more than 100ms with small crowd sizes. For the Base and the Small Query stages, the stopping size was just 5 clients², and for the Large Object stage the stopping size was 25 clients. These results indicate that the server is poorly provisioned in general, with bandwidth being provisioned better than the rest of the infrastructure. The site administrators confirmed that MFC experiments provided an accurate view of the server configuration, as it is not provisioned to serve a large volume of requests (since it hosts a relatively small number of pages). The server logs indicate that the MFC requests arrived within a maximum 1s of one another.

Univ-2: We measured the Univ-2 Web server at three different times on Oct 5, 2007, using MFC-mr with a 250ms threshold (after discussions with the operators). The Univ-2 server runs Apache version 2 and is behind a 1Gbps link, with a relatively small amount of background traffic (maximum of 4.2 requests/sec in the morning experiment). The relative volume of background (vs. MFC requests) traffic was 67%, 52%, 59% for the three experiments.

From the results in Table 3(a), we see that there are a few cases in which MFC did not result in a 250ms response time degradation, even when using all available clients. However, in these cases we noticed that as soon as the number of simultaneous requests crossed 130, the MFC caused a 150-200ms increase in the base response time. With additional clients, it is likely that the response

Expt details	Base		Small Qry		Large Obj		MFC	Other
	Time (Span)	Crowd size	Time (Span)	Crowd size	Time (Span)	Crowd size	Traffic (% of all)	Traffic reqs/s
MFC-mr 250ms	1015 (557s)	NoStop (140)	1025 (441s)	130	1035 (330s)	110	2682 (33%)	4.2
MFC-mr 250ms	1725 (378s)	150	1740 (382s)	130	1755 (300s)	NoStop (150)	2829 (48%)	2.9
MFC-mr 250ms	2354 (295s)	NoStop (150)	0000 (354s)	130	0010 (323s)	110	2442 (41%)	3.5

(a) Univ-2

Expt details	Base		Small Qry		Large Obj		MFC	Other
	Time (Span)	Crowd size	Time (Span)	Crowd size	Time (Span)	Crowd size	Traffic (% of all)	Traffic reqs/s
MFC-mr 250ms	0925 (303s)	90	0935 (176s)	30	0950 (318s)	NoStop (150)	1388 (7.9%)	20.3
MFC-mr 250ms	1605 (330s)	110	1620 (173s)	30	1630 (285s)	NoStop (130)	1422 (49%)	18.7
MFC-mr 250ms	2255 (299s)	NoStop (150)	2305 (171s)	30	2320 (308s)	NoStop (150)	1543 (13.7%)	12.5

(b) Univ-3

Table 3: Results for Univ-2 and Univ-3. The day-time experiments were run on Oct 5th, and late evening experiments on Oct 6th. For each experiment we indicate when it was started and how long it ran. Times are in US CDT.

time increase would have crossed the set threshold.

An interesting observation from these experiments is that, irrespective of the MFC stage, the experiments seem to consistently stop (or show a substantial degradation in response time) for crowds of sizes 110-150. This holds even for the large object stage, which is surprising because the server's access bandwidth is very well provisioned. Software configuration artifacts (e.g., limits on the number of server threads) or buffer limitations might explain these observations. As we discussed in Section 3, an observed increase in response time may not always be due to server resource constraints, but rather due to artifacts such as server-side request scheduling, resource serialization, or buffer exhaustion.

The Univ-2 administrators agreed that software configuration may be the reason (though they did not know the exact reason). The server's software configuration had not changed in several years, and the operators requested us to run additional MFC experiments against a new configuration with a much larger bound on the number of threads. The administrators felt that the MFC approach could prove useful to tune both the hardware and software configuration of their server.

Univ-3: We conducted similar experiments on the Univ-3 Web site on October 5, 2007. The Web site runs on a 1.5GHz Sun V240 server. Compared to Univ-2, the rate of background traffic at the Univ-3 server was 5X to 9X higher. The highest rate was observed during the morning experiment (20 requests/s) and the lowest rate observed during the late evening experiment (12.5 requests/s).

We see in Table 3(b) that the base HTTP processing capabilities are adequate and comparable to Univ-2. The Large Object stage shows no response time impact, confirming that the bandwidth was well-provisioned. The site's ability to handle the small query request was poor, however, as the response time showed a significant increase with just 30 simultaneous requests.

For Univ-3, we also observe some effects due to the variations in background traffic. For instance, in the morning and afternoon Base stage experiments, the stopping crowd size is lower, i.e., when there was more background traffic. The late evening experiment (22:55 hrs) did not cause the response time to increase beyond the

threshold. Background traffic has little impact on the Small Query and Large Object stages as the results are similar for all three experiments. Small Query results are affected solely by the constrained query handling capacity, while the Large Object stage results are influenced primarily by the abundant bandwidth. To gain a thorough understanding of the limitations of a server's resources, it may be useful to run MFCs at under diverse background traffic conditions.

Upon examining the results, the Univ-3 site operators echoed the sentiments of the operators of QTP, Univ-1, and Univ-2 sites, namely that a diagnostic tool like MFC that is capable of providing guidelines on individual resource bottlenecks is useful. In their experience, site provisioning was often based on best guesses followed by reactive changes. The operators also mentioned that MFC was non-intrusive as the impact on their servers was minimal. Comparisons between Head and Large Object results were of particular value to them; they felt that they could have used the results to debug a recent incident in which a large number of simultaneous downloads of a popular video frustrated another user downloading a different large file. It was unclear if the poor performance of the frustrated user was due to a bandwidth bottleneck or request handling constraints. Since Base showed a discernible response time increase while Large Object did not, they felt the real problem was more likely in request handling, rather than bandwidth provisioning. The Small Query experiment helped them to recall that their legacy infrastructure was not caching responses appropriate thus causing the perceptible degradation even with small crowd sizes.

Our experiments with cooperating production Web sites proved to be quite useful to demonstrate the practical benefits of MFCs. Site operators agreed that the technique did not impose a significant overhead. We also found that inferences based on MFC experiments confirmed suspected issues, and in some cases revealed new information or brought provisioning/configuration issues to the attention of site operators. Having access to server logs and other data also allowed us to examine issues such as synchronization and the effects of background traffic more directly.

5 Large-Scale Measurements

The MFC approach is non-intrusive, automatic, and tunable to the specific content on a server. Thus, MFCs can be run against servers “in the wild” to get insights into resource provisioning without disrupting the servers’ functioning. In August and September 2007, we measured several hundred Web servers, about 90 known phishing servers, and a few hundred servers of startup companies. We used the standard version of MFC with a threshold of 100ms. Our experiments required a minimum of 50 client nodes; the maximum number of clients we used depended on how many PlanetLab nodes were responsive during a given experiment (maximum was 85). Each participating client sent at most one request (i.e., we do not use MFC-mr).

5.1 Generic Web servers

We first present results of running the MFC Base stage against more than 400 Web servers. These were selected from a list of Web site rankings maintained by Quantcast [19]. Servers with different levels of popularity are well represented in our selection. We selected 114 sites ranked 1-1K, 107 sites ranked 1K-10K, 118 sites ranked 10K-100K, and 148 sites ranked 100K-1M. We expect that servers with a smaller rank (more popular Web servers) would be qualitatively similar to one another in terms of provisioning, and, more importantly, better-provisioned than Web servers with much larger ranks.

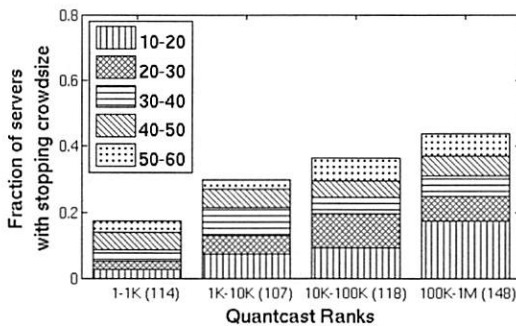


Figure 7: Breakdown of stopping crowd sizes for the Base stage for various Quantcast rank ranges.

Figure 7 shows a summary of the crowd size at which different servers showed more than 100ms increase in the median response time for the Base stage of the MFC. For each rank category, we also break down the stopping crowd sizes into sub-ranges as shown. As expected, the total fraction of servers that show a 100ms increase in response time increases steadily as we move to servers with larger rank indexes (17% for the 1-1K category vs

45% for 100K-1M category). More than 15% of the servers in the largest rank category (100K-1M) can handle at most 20 simultaneous HEAD requests before the response time increases visibly. Surprisingly, we also find that $\sim 10\%$ of the Web sites in the 1-1K rank category degrade with less than 40 simultaneous requests.

Next, for the Small Query stage we selected around 400 Web servers, each hosting at least one object that fits our definition of Small Query (see Section 2.2). All clients requested the same object at the target server. We measured 106, 103, 103, and 122 servers in the four rank ranges in ascending order of the ranks. The measurements summarized in Figure 8 show that the provisioning of the servers is strongly correlated with popularity: the fraction of servers which show ≥ 100 ms degradation in response time increases significantly as the server popularity decreases. Comparing Figures 8 and 7, we see that across all the rank-ranges a much larger fraction of servers showed more than a 100ms degradation in response time with the Small Query stage compared with the Base stage. Among the servers with largest ranks (100K-1M), about 75% of the servers cannot handle more 50 simultaneous queries and about 45% cannot handle more than 20 simultaneous queries (these numbers were 38% and 18%, respectively, for the Base stage). Somewhat surprisingly, even among the highest-ranked servers, about 20% cannot handle any more than 40 simultaneous queries.

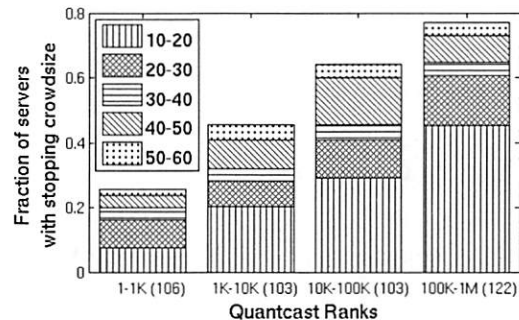


Figure 8: Breakdown of stopping crowd sizes for the Small Query stage for various Quantcast rank ranges.

A possible reason for the difference between the Base and Small Query is that the latter typically requires more processing or more accesses to other back-end services (e.g., databases) compared to the lightweight HEAD requests of the Base stage. With a more resource-intensive workload, we note a larger fraction of the Web sites showing a degradation.

For the Large Object stage, we measured 129, 100, 114, and 103 servers in the four rank ranges, where each server hosts at least one Large Object (size between

100KB and 2MB). The bandwidth provisioning of Web servers appears less correlated with the server's popularity than the provisioning of the back-end database modules (see Figure 8) or the basic HTTP processing (see Figure 7). Except for the most popular servers (1-1K), about 45-55% of the servers in the rest of the categories cannot handle more than 50 simultaneous requests. Also, for the first two categories with the most popular servers (1-1K and 1K-10K), the fractions are similar across the Small Query and the Large Object stages. But, for the remaining two categories, a much smaller fraction of servers exhibit a degradation in response time during the Large Object stage (57% and 55%) when compared to Small Queries (65% and 77%). Most popular servers provision both their access bandwidth and back-end database and processing quite well. Lower rung servers appear to provision their bandwidth relatively better than their back-end data processing capability.

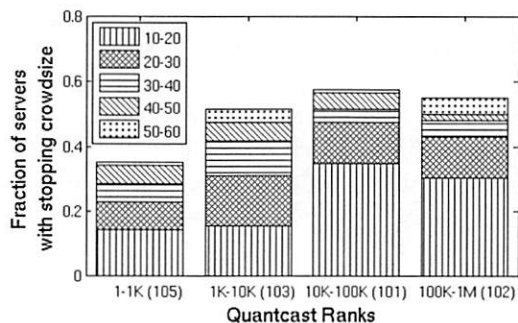


Figure 9: Breakdown of stopping crowd sizes for the Large Object stage for various Quantcast rank ranges.

5.2 Startup Companies

We also measured Web servers of a few hundred startup companies. Startup sites could benefit from a MFC approach, since they are often ill-prepared for a “success disaster” resulting from coverage in popular news sites or technology blogs. We compiled a list of recent startups from technology blogs. We expected that most of them would be deployed at commercial Web hosting services, and thus, likely to be reasonably well-provisioned.

Stopping Crowdsize	Percentage of servers	
	Base	Small Query
10-20	24%	33%
20-30	6%	12%
30-40	7%	6%
40-50	6%	5%
No-Stop	58%	44%

Table 4: Stopping crowd sizes for startup servers

We ran the Base MFC stage on 107 startup servers; results are shown in Table 4. About 68 servers (58%)

did not show a visible degradation in response time even with 50 requests. At the other extreme, 24% of servers showed a 100ms degradation with ≤ 20 requests. A lot fewer servers showed degradation in response times for the intermediate crowd sizes. The results for the Large Object stage (we ran against 103 servers) were qualitatively similar to the Base stage, with 30% of the servers showing a 100ms increase with a crowd size less than 30 (not shown). Compared to the Base and Large Object stages, the Small Query experiment showed a marginally greater degradation in the performance of the startup sites (we measured 82 servers). Around 33% of the tested servers were unable to handle more than 20 requests in the Small Query experiment, and around 56% of the sites stopped with a crowd size less than 50 (Table 4). Overall, we find that a significant fraction of startup servers (between 24% and 33%) cannot handle more than a handful (≤ 20) of requests, and hence are ill-prepared for even low-volume request floods.

5.3 Phishing Sites

We also conducted a measurement study of 89 phishing sites obtained from Phish-tank [18] where we ran the Base MFC stage.

Stopping Crowdsize	Percentage of servers
10-20	12%
20-30	16%
30-40	11%
40-50	11%
No-Stop	50%

Table 5: Stopping crowd sizes for HEAD request for phishing servers

Our intuition was that a very small fraction of phishing sites would be hosted at well-provisioned Web hosting service providers. Thus, we expected phishers to be similar, if not worse, compared to servers in the Quantcast 100K-1M rank range. From Table 5, we see that a significant fraction (28%) cannot handle more than 30 requests. For servers in the 100K-1M rank category (Figure 7), we find that the corresponding fraction of sites was 18%, suggesting that most of the phishing sites are hosted on fairly low-end servers similar to the 100K-1M ranked Web sites. Table 5 also shows that about 50% of the phishing sites did not show a 100ms increase in response time even with a crowd of 50 clients—the corresponding fraction for servers in the 100K-1M category, 62%, is only slightly higher (Table 5). Indeed, the distribution of the request handling capabilities of the phishing sites is quite similar to low-end Web sites.

6 Extensions

Role of “Measurers”: We can augment MFC with a set of “measurers” that independently measure the response time at the target and report these to the coordinator. The measurers can either make concurrent requests for the same object requested by the crowd or have the flexibility to request other objects. The latter approach can help quantify correlations among resources on the target server (e.g., how does a disk-intensive workload impact the response time of a database-intensive request?).

DDoS Vulnerabilities: Web sites can be the targets of either network-level or application-level DDoS attacks. The former class targets a server’s incoming (e.g., SYN-floods) or outgoing bandwidth (e.g., e-protests) and the latter targets server’s CPU, memory, disk or the back-end database. The wide-spread use of botnets increases the risk of carefully crafted application-level DDoS attacks. Solution proposals for network-level attacks include capabilities [25, 17] and bandwidth amplifications [5, 11], while resource payment mechanisms [21] have been proposed for application-level attacks. A site operator must first understand which resources are the most easily vulnerable to attacks. Second, the effectiveness of some protection mechanisms depends on the volume of the attack traffic (e.g., [21, 17]). Thus, the operator needs to understand at what volume of requests a server resource starts to “keel over”. For instance, if a server’s response time does not increase during the Large Object stage for very large crowd sizes, but does so at a small crowd size for the Small Query workload, then the server is highly vulnerable to even the most simple application-level attacks on the back-end data processing subsystem. By comparing inferences drawn from the different stages of a MFC the operator can address these issues.

Staggered MFC: MFC enforces tight synchronization of requests by scheduling them to arrive at the target simultaneously. However, if a Web server performs poorly with respect to tight synchronization, but provides low response times when the requests arrive somewhat staggered, then we can conclude that the server can handle the medium and low volume flash-crowds reasonably well. Operators can benefit from understanding how the application infrastructure behaves when the request inter-arrival times follow a certain distribution. A simple extension to MFC can provide this capability – the coordinator schedules the clients such that the target sees 1 request every m milliseconds. Other non-uniform distributions of inter-arrival times are also easy to implement.

7 Related Work

Web server Benchmarking: Benchmarking tools [20, 24, 22, 13] can emulate multiple user sessions, create

client requests for dynamic content, and model standard workloads for banking, e-commerce, and Web browsing. These benchmarks provide controlled emulation of the client side behavior in a lab setting (clients and server on the same LAN). These tools stress test a server by changing parameters such as the number of active clients and the inter-arrival times of requests. In contrast, MFC uses clients that are distributed across the wide-area network, providing the ability to understand server performance under realistic networking conditions. Also, MFC provides the ability to exercise specific server resources in a controlled fashion, yielding detailed observations.

Measuring and Modeling Flash Crowds: There are several proposals for modeling flash crowd events [2, 23, 6]. One such technique [2] uses real flash crowd traces to model request patterns and inter-arrival times to study the effectiveness of various caching techniques during flash crowd situations. Such approaches only capture coarse-grained behavioral characteristics of flash-crowd events, and as such do not address more fine-grained details of how the requests impact individual server components. By devising specific request types MFC provides the ability to capture these fine-grained aspects as well.

There are also techniques that study differences in client request patterns to distinguish legitimate flash crowds from malicious DDoS attacks [9]. Another technique [6] identifies flash crowds by examining performance degradation in responses. Such techniques primarily act as diagnostic aids when a server is experiencing extreme load. In contrast, MFC operates in a significantly lower load regime while providing the ability to identify the request volume at which a server’s response time begins to show perceptible degradation.

Software Profiling: Several efforts attempt to profile hosts to reveal interesting aspects of software platforms or software artifacts that can indicate bottlenecks. TBIT [16] fingerprints the TCP versions (e.g., Reno, Newreno or SACK) used on servers. Controlled lab experiments have been used to evaluate the robustness of different TCP/IP implementations [7]. NMAP [14] and p0f [15] are commonly used to remotely fingerprint operating systems running on network-connected hosts. MFC can complement these tools to provide a more comprehensive resource profile of Internet servers.

Researchers have suggested techniques for inferring performance bottlenecks in distributed systems as events or requests flow through the system [4, 1]. Although these approaches treat the target system as a black box on the whole, they require non-trivial instrumentation of the target to shed light on why some requests are delayed more than others. MFC is designed to provide useful inferences even without any involvement from or instrumentation of the target server. By additionally instrumenting the target, we can improve the accuracy of our

inferences.

Commercial Services: Keynote [10] provides wide-area measurement services, with a focus on end-user experience. They use response times for single requests measured from a global network of computers to infer resource bottlenecks but they do not synchronize requests in any way. In contrast, MFC can infer resource bottlenecks that are not visible when the web server processes a single request, but surface only under more intensive synchronized loads.

8 Summary

We have presented the design, implementation, and evaluation of mini-flash crowds (MFC)—a light-weight, non-intrusive, wide-area profiling service for revealing resource bottlenecks in a Web server infrastructure. Through controlled measurements with an increasing number of clients making synchronized requests to exercise specific resources of a remote server, we are able to faithfully track the impact on different server resources. We performed extensive validation experiments to verify that our approach can offer useful and accurate information regarding resource provisioning. We conducted several tests on co-operating Web sites, including one large commercial site, which showed that our approach is practical and safe. The operators of the cooperating sites confirmed the inferences we made and found our observations regarding the provisioning of their server infrastructure quite useful. We ran MFC against hundreds of servers of differing grades of popularity and with correspondingly different server infrastructures. These measurements indicate that back-end processing is a key bottleneck for many servers of medium to low popularity and that the access bandwidth is less constrained overall. We are currently exploring numerous extensions to MFC including tailored use for specific sites and using workloads with specific distributions of inter-arrival times.

References

- [1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proc. ACM SOSP* (2003).
- [2] ARI, I., HONG, B., MILLER, E., BRANDT, S., AND LONG, D. Managing of Flash Crowds on the Internet. In *Proc. of MASCOTS* (2003).
- [3] Anti-Web HTTPD Homepage. <http://www.hcs.w.org/awhttpd/>.
- [4] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *Proc. of OSDI* (2004), pp. 259–272.
- [5] CASADO, M., AKELLA, A., CAO, P., PROVOS, N., AND SHENKER, S. Cookies Along Trust Boundaries (CAT): Accurate and Deployable Flood Protection. In *Proc. of USENIX SRUTI* (San Jose, CA, July 2006).
- [6] CHEN, X., AND HEIDEMANN, J. Flash Crowd Mitigation via Adaptive Admission Control Based on Application Level Observations. *Transactions on Information Technology* 5, 3 (2005), 532–569.
- [7] GAO, T., AND MAHDAVI, J. On Current TCP/IP implementations and Performance. Unpublished manuscript, Aug. 2000.
- [8] HEINLEIN, P. Fastcgi: A high-performance gateway interface. In *Proc. of WWW* (May 1996).
- [9] JUNG, J., KRISHNAMURTHY, B., AND RABINOVICH, M. Flash crowds and denial of service attacks: Characterization, and implications for CDNs and web sites. In *Proc. of WWW* (2002).
- [10] KEYNOTE SYSTEMS, INC. <http://www.keynote.com>, June 2003.
- [11] MAHIMKAR, A., DANGE, J., SHMATIKOV, V., VIN, H., AND ZHANG, Y. dFence: Transparent Network-based Denial of Service Mitigation. In *Proc. of NSDI* (2007).
- [12] Mongrel: V2. <http://mongrel.rubyforge.org>.
- [13] MOSBERGER, D., AND JIN, T. httpf: A Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev.* 26, 3 (1998), 31–37.
- [14] Nmap: Free security scanner for network exploration. <http://insecure.org/nmap/>.
- [15] p0f: A passive finger printing tool. <http://lcamtuf.coredump.cx/p0f.shtml>.
- [16] PADHYE, J., AND FLOYD, S. Identifying the TCP Behavior of Web Servers. In *ACM SIGCOMM* (Aug. 2001).
- [17] PARNO, B., WENDLANDT, D., SHI, E., PERRIG, A., MAGGS, B., AND HU, Y.-C. Portcullis: Protecting Connection Setup from Denial-of-Capability Attacks. *SIGCOMM Comput. Commun. Rev.* 37, 4 (2007), 289–300.
- [18] Phishtank. <http://phishtank.org>.
- [19] Quantcast: Open Internet Ratings Service. <http://www.quantcast.com/>.
- [20] SPECweb2005 Benchmark. <http://www.spec.org/web2005>, 1999.
- [21] WALFISH, M., VUTUKURU, M., BALAKRISHNAN, H., KARGER, D., AND SHENKER, S. DDoS Defense by Offense. In *Proc. of ACM SIGCOMM* (September 2006).
- [22] Mindcraft Benchmarks: WebStone. <http://www.mindcraft.com/webstone/>.
- [23] WEI, S., AND MIRKOVIC, J. A Realistic Simulation of Internet Scale Events. In *Proc. of VALUETOOLS* (2006).
- [24] SPECweb99 Benchmark. <http://www.spec.org/osg/web99>, 1999.
- [25] YANG, X., WETHERALL, D., AND ANDERSON, T. A DoS-limiting Network Architecture. In *Proc. of SIGCOMM* (2005).
- [26] ZHANG, Y., DUFFIELD, N., PAXSON, V., AND SHENKER, S. On the Constancy of Internet Path Properties. In *Proc. of IMW* (Nov. 2001).

Notes

¹FastCGI forks a new process for each request. As the number of requests increases, each of the forked process independently inherits the memory image of the parent process leading to very high memory usage during the experiment.

²As indicated in Figure 2, the experiment runs until the crowd size reaches 15. We analyzed the results to identify the earliest crowd size at which a 100ms increase occurs.

A Dollar from 15 Cents: Cross-Platform Management for Internet Services

Christopher Stewart[†] Terence Kelly* Alex Zhang* Kai Shen[†]

[†]University of Rochester *Hewlett-Packard Laboratories

Abstract

As Internet services become ubiquitous, the selection and management of diverse server platforms now affects the bottom line of almost every firm in every industry. Ideally, such cross-platform management would yield high performance at low cost, but in practice, the performance consequences of such decisions are often hard to predict. In this paper, we present an approach to guide cross-platform management for real-world Internet services. Our approach is driven by a novel performance model that predicts application-level performance across changes in platform parameters, such as processor cache sizes, processor speeds, etc., and can be calibrated with data commonly available in today's production environments. Our model is structured as a composition of several empirically observed, parsimonious sub-models. These sub-models have few free parameters and can be calibrated with lightweight passive observations on a current production platform. We demonstrate the usefulness of our cross-platform model in two management problems. First, our model provides accurate performance predictions when selecting the next generation of processors to enter a server farm. Second, our model can guide platform-aware load balancing across heterogeneous server farms.

1 Introduction

In recent years, Internet services have become an indispensable component of customer-facing websites and enterprise applications. Their increased popularity has prompted a surge in the size and heterogeneity of the server clusters that support them. Nowadays, the management of heterogeneous server platforms affects the bottom line of almost every firm in every industry. For example, purchasing the right server makes and models can improve application-level performance while reducing cluster-wide power consumption. Such management decisions often span many server platforms that, in practice, cannot be tested exhaustively. Consequently, cross-platform management for Internet services has historically been ad-hoc and unprincipled.

Recent research [9,14,31,37,40,41,44] has shown that performance models can aid the management of Internet services by predicting the performance consequences of contemplated actions. However, past models for Internet services have not considered platform configurations such as processor cache sizes, the number of processors, and processor speed. The effects of such parameters are fundamentally hard to predict, even when data can be collected by any means. The effects are even harder to predict in real-world production environments, where data collection is restricted to passive measurements of the running system.

This paper presents a cross-platform performance model for Internet services, and demonstrates its use in making management decisions. Our model predicts application-level response times and throughput from a composition of several sub-models, each of which describes a measure of the processor's performance (henceforth, a processor metric) as a function of a system parameter. For example, one of our sub-models relates cache misses (a processor metric) to cache size (a system parameter). The functional forms of our sub-models are determined from empirical observations across several Internet services and are justified by reasoning about the underlying design of Internet services. Our knowledgeable sub-models are called *trait models* because, like human personality traits, they stem from empirical observations of system behaviors and they characterize only one aspect of a complex system. Figure 1 illustrates the design of our cross-platform model.

The applicability of our model in real-world production environments was an important design consideration. We embrace the philosophy of George Box, "All models are wrong, but some [hopefully ours] are useful." [10] To reach a broad user base, our model targets third-party consultants. Consultants are often expected to propose good management decisions without *touching* their clients' production applications. Such inconvenient but realistic restrictions forbid source code instrumentation and controlled benchmarking. In many ways the challenge in such data-impovertised environments fits the words of the old adage, "trying to make a dollar from 15 cents." Typically, consultants must make do with data available from standard monitoring utilities and

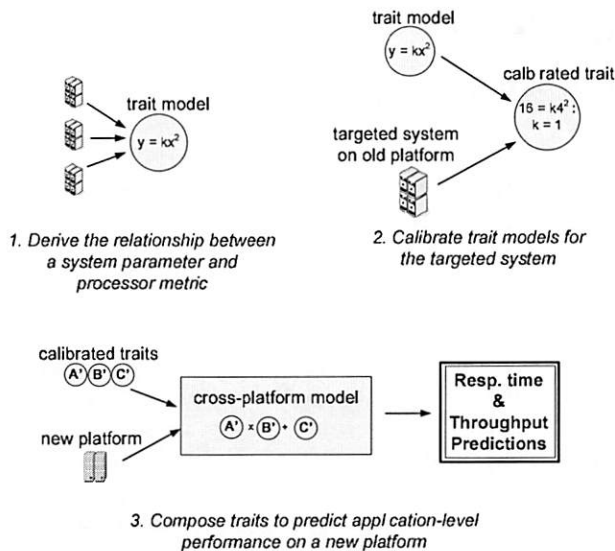


Figure 1: The design of our cross-platform performance model. The variable x represents a system parameter, *e.g.*, number of processors or L1 cache size. The variable y represents a processor metric (*i.e.*, a measure of processor performance), such as instruction count or L1 cache misses. Application-level performance refers to response time and throughput, collectively.

application-level logs. The simplicity of our sub-models allows us to calibrate our model using only such readily available data.

We demonstrate the usefulness of our model in the selection of high-performance, low-power server platforms. Specifically, we used our model (and processor power specifications) to identify platforms with high performance-per-watt ratios. Our model outperforms alternative techniques that are commonly used to guide platform selection in practice today. We also show that model-driven load balancing for heterogeneous clusters can improve response times. Under this policy, request types are routed to the hardware platform that our model predicts is best suited to their resource requirements.

The contributions of this paper are:

1. We observe and justify trait models across several Internet services.
2. We integrate our trait models into a cross-platform model of application-level performance.
3. We demonstrate the usefulness of our cross-platform model for platform selection and load balancing in a heterogeneous server farm.

The remainder of this paper is organized as follows. Section 2 overviews the software architecture, processing patterns, and deployment environment of the realistic Internet services that we target. Section 3 presents several trait models. Section 4 shows how we compose trait

models into an established framework to achieve an accurate cross-platform performance prediction and compares our approach with several alternatives. Section 5 shows how trait-based performance models can guide the selection of server platforms and guide load balancing in a heterogeneous server cluster. Section 6 reviews related work and Section 7 concludes.

2 Background

Internet services are often designed according to a three-tier software architecture. A response to an end-user request may traverse software on all three tiers. The first tier translates end-user markup languages into and out of business data structures. The second tier (*a.k.a.* the business-logic tier) performs computation on business data structures. Our work focuses on this tier, so we will provide a detailed example of its operation below. The third tier provides read/write storage for abstract business objects. Requests traverse tiers via synchronous communication over local area networks (rather than shared memory) and a single request may revisit tiers many times [26]. Previous studies provide more information on multi-tier software architectures [29, 44].

Business-logic computations are often the bottleneck for Internet services. As an example, consider the business logic of an auction site: computing the list of currently winning bids can require complex considerations of bidder histories, seller preferences, and shipping distances between bidders and sellers. Such workloads are processor intensive, and their effect on application-level performance depends on the underlying platform's configuration in terms of cache size, on-chip cores, hyperthreading, etc. Therefore, our model, which spans a wide range of platform configurations, naturally targets the business-logic tier. Admittedly, application-level performance for Internet services can also be affected by disk and network workloads at other tiers. Previous works [14, 41] have addressed some of these issues, and we believe our model can be integrated with such works.

Internet services keep response times low to satisfy end-users. However as requests arrive concurrently, response times increase due to queuing delays. In production environments, resources are adequately provisioned to limit the impact of queuing. Previous analyses of several real enterprise applications [40] showed max CPU utilizations below 60% and average utilizations below 25%. Similar observations were made in [8]. Services are qualitatively different when resources are adequately provisioned compared to overload conditions. For example, contention for shared resources is more pronounced under overload conditions [29].

Time stamp	Type 1	Aggregate counts			L1 miss	L2 miss
		Type 2	...	Instr. count (x10 ⁴)		
2:00pm	18	42	...	280322	31026	2072
2:01pm	33	36	...	311641	33375	2700

Table 1: Example of data available as input to our model. Oprofile [2] collects instruction counts and cache misses. Apache logs [1] supply frequencies of request types.

2.1 Nonstationarity

End-user requests can typically be grouped into a small number of types. For example, an auction site may support request types such as bid for item, sell item, and browse items. Requests of the same type often follow similar code paths for processing, and as a result, requests of the same type are likely to place similar demands on the processor. A request mix describes the proportion of end-user requests of each type.

Request mix nonstationarity describes a common phenomenon in real Internet services: the relative frequencies of request types fluctuate over long and short intervals [40]. Over a long period, an Internet service will see a wide and highly variable range of request mixes. On the downside, nonstationarity requires performance models to generalize to previously unseen transaction mixes. On the upside, nonstationarity ensures that observations over a long period will include more unique request mixes than request types. This diversity over-constrains linear models that consider the effects of each request type on a corresponding output metric (*e.g.*, instruction count), which enables parameter estimation using regression techniques like Ordinary Least Squares.

2.2 Data Availability

In production environments, system managers must cope with the practical (and often political) issues of trust and risk during data collection. For example, third-party consultants—a major constituent of our work—are typically seen as semi-trusted decision makers, so they are not allowed to perform controlled experiments that could pose availability or security risks to business-critical production systems. Similarly, managers of shared hosting centers are bound by business agreements that prevent them from accessing or modifying a service’s source code. Even trusted developers of the service often relinquish their ability to perform invasive operations that could cause data corruption.

Our model uses data commonly available in most production environments, even in consulting scenarios. Specifically, we restrict our model inputs to logs of request arrivals and CPU performance counters. Table 1 provides an example of our model’s inputs. Our model

also uses information available from standard platform specification sheets such as the number of processors and on-chip cache sizes [6].

3 Trait Models

Trait models characterize the relationship between a system parameter and a processor metric. Like personality traits, they reflect one aspect of system behavior (*e.g.*, sensitivity to small cache sizes or reaction to changes in request mix). The intentional simplicity of trait models has two benefits for our cross-platform model. First, we can extract parsimonious yet general functional forms from empirical observations of the parameter and output metric. Second, we can automatically calibrate trait models with data commonly available in production environments.

Our trait models take the simplest functional form that yields low prediction error on the targeted system parameter and processor metric. We employ two sanity checks to ensure that our traits reflect authentic relationships—not just peculiarities in one particular application. First, we empirically validate our trait models across several applications. Second, we justify the functional form of our trait models by reasoning about the underlying structure of Internet services.

In this section, we observe two traits in the business logic of Internet services. First, we observe that a power law characterizes the miss rate for on-chip processor caches. Specifically, data-cache misses plotted against cache size on a log-log scale are well fit by a linear model. We justify such a heavy-tail relationship by reasoning about the memory access patterns of background system activities. Compared to alternative functional forms, a power law relationship achieves excellent prediction accuracy with few free model parameters.

Second, we observe that a linear request-mix model describes instruction count and aggregate cache misses. This trait captures the intuition that request type and volume are the primary determinants of runtime code paths. Our experiments demonstrate the resiliency of request mix models under a variety of processor configurations. Specifically, request-mix models remain accurate under SMP, multi-core, and hyperthreading processors.

3.1 Error Metric

The normalized residual error is the metric that we use to evaluate trait models. We also use it in the validation of our full cross-platform model. Let Y and \hat{Y} represent observed and predicted values of the targeted output metric, respectively. The residual error, $E = Y - \hat{Y}$ tends toward zero for good models. The normalized residual error, $\frac{|E|}{Y}$, accounts for differences in the magnitude of Y .

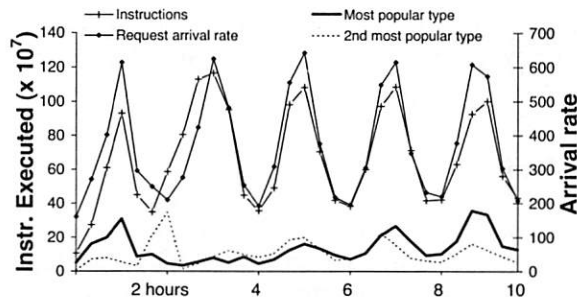


Figure 2: Nonstationarity during a RUBiS experiment. Request arrivals fluctuate in a sinusoidal fashion, which correspondingly affects the aggregate instructions executed. The ratio of the most popular request type to the second most popular type (i.e., $\frac{freq_{most}}{freq_{2nd}}$) ranges from 0.13 to 12.5. Throughout this paper, a request mix captures per-type frequencies over a 30 second interval.

3.2 Testbed Applications

We study the business logic of three benchmark Internet Services. RUBiS is a J2EE application that captures the core functionalities of an online auction site [3]. The site supports 22 request types including browsing for items, placing bids, and viewing a user's bid history. The software architecture follows a three-tier model containing a front-end web server, a back-end database, and Java business-logic components. The StockOnline stock trading application [4] supports six request types. End users can buy and sell stocks, view prices, view holdings, update account information, and create new accounts. StockOnline also follows a three-tier software architecture with Java business-logic components. TPC-W simulates the activities of a transactional e-commerce bookstore. It supports 13 request types including searching for books, customer registration, and administrative price updates. Applications run on the JBoss 4.0.2 application server. The database back-end is MySQL 4.0. All applications run on the Linux 2.6.18 kernel.

The workload generators bundled with RUBiS, StockOnline, and TPC-W produce synthetic end-user requests according to fixed long-term probabilities for each request type. Our previous work showed that the resulting request mixes are qualitatively unlike the nonstationary workloads found in real production environments [40]. In this work, we used a nonstationary sequence of integers to produce a nonstationary request trace for each benchmark application. We replayed the trace in an open-arrival fashion in which the aggregate arrival rate fluctuated. Figure 2 depicts fluctuations in the aggregate arrival rate and in the relative frequencies of transaction types during a RUBiS experiment. Our nonstationary sequence of integers is publicly available [5] and can be used to produce nonstationary mixes for any application with well-defined request types.

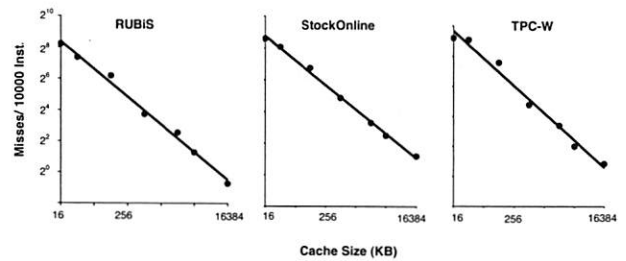


Figure 3: Cache misses (per 10k instructions) plotted against cache size on a log-log plot. Measurements were taken from real Intel servers using the Oprofile monitoring tool [2]. The same nonstationary workload was issued for each test. Cache lines were 64 bytes.

3.3 Trait Model of Cache Size on Cache Misses

Figure 3 plots data-cache miss rates relative to cache size on a log-log scale. Using least squares regression, we calibrated linear models of the form $\ln(Y) = B\ln(X) + A$. We observe low residual errors for each application in our testbed. Specifically, the largest normalized residual error observed for RUBiS, StockOnline, and TPCW is 0.08, 0.03, and 0.09 respectively. The calibrated B parameters for RUBiS, StockOnline, and TPCW are -0.83, -0.77, and -0.89 respectively. Log-log linear models with slopes between $(-2, 0)$ are known as *power law distributions* [19, 33, 43]

We justify our power-law trait model by making observations on its rate of change, shown in Equation 1.

$$\frac{dY}{dX} = Be^A X^{B-1} \quad (1)$$

For small values of X , the rate of change is steep, but as X tends toward infinity the rate of change decreases and slowly (i.e., with a heavy tail) approaches zero. The heavy-tail aspect of a power law means the rate of change decreases more slowly than can be described using an exponential model. In terms of cache misses, this means a power-law cache model predicts significant miss rate reductions when *small* caches are made larger, but almost no reductions when *large* caches are made larger. The business logic tier for Internet services exhibits such behavior by design. Specifically, per-request misses due to lack of capacity are significantly reduced by larger L1 caches. However, garbage collection and other infrequent-yet-intensive operations will likely incur misses even under large cache sizes.

A power law relationship requires the calibration of only two free parameters (i.e., A , and B), which makes it practical for real-world production environments. However, there are many other functional forms that have only two free parameters; how does our trait model compare to alternatives? Table 2 compares logarithmic,

		Log	Exp.	Power	Log
				law	-normal
RUBiS	lowest	0.011	0.105	0.005	0.001
	median	0.094	0.141	0.028	0.027
	highest	0.168	0.254	0.080	0.072
Stock	lowest	0.013	0.010	0.010	0.012
	median	0.075	0.099	0.023	0.024
	highest	0.026	0.142	0.034	0.042
TPCW	lowest	0.046	0.044	0.011	0.007
	median	0.109	0.084	0.059	0.060
	highest	0.312	0.146	0.099	0.101

Table 2: Normalized residual error of cache models that have fewer than three free parameters. The lowest, median, and highest normalized residuals are reported from observations on seven different cache sizes.

exponential, and power law functional forms. Power law models have lower median normalized residual error than logarithmic and exponential models for each application in our testbed. Also, we compare against a generalized (quadratic) log-normal model, $\ln(Y) = B_2 \ln(X)^2 + B_1 \ln(X) + A$. This model allows for an additional free parameter (B_2) in calibration, and is expected to provide a better fit, though it cannot be calibrated from observations on one machine. Our results show that the additional parameter does not substantially reduce residual error. For instance, the median residual error for the power law distribution is approximately equal to that of the generalized log-normal distribution. We note that other complex combinations of these models may provide better fits, such as Weibull or power law with exponential cut-off. However, such models are hard to calibrate with the limited data available in production environments.

3.4 Trait Models of Request Mix on Instructions and Cache Misses

Figure 4 plots a linear combination of request type frequencies against the instruction count, L1 misses, and L2 misses for RUBiS. Our parsimonious linear combination has only one model parameter for each request type, as shown below.

$$C_k = \sum_{\text{types } j} \alpha_{jk} N_j \quad (2)$$

Where C_k represents one of the targeted processor metrics and N_j represents the frequency of requests of type j . The model parameter α_{jk} transforms request-type frequencies into demand for processor resources. Intuitively, α_{jk} represents the typical demand for processor resource k of type j . We refer to this as a request-mix model, and we observe excellent prediction accuracy. The 90th percentile normalized error for instructions, L1 misses, and L2 misses were 0.09, 0.10, 0.06 respectively.

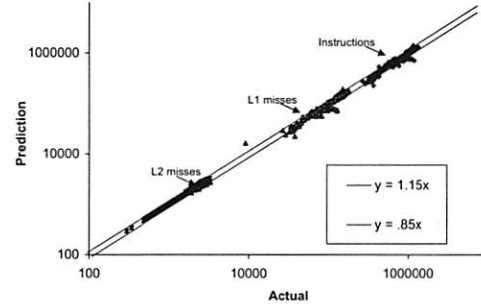


Figure 4: RUBiS instruction count and aggregate cache misses (L1 and L2) plotted against a linear request mix model's predictions. Measurements were taken from a single processor Intel Pentium D server. Lines indicate 15% error from the actual value. Collected with OProfile [2] at sampling rate of 24000.

Request-mix models are justifiable, because business-logic requests of the same type typically follow similar code paths. The number of instructions required by a request will likely depend on its code path. Similarly, cold-start compulsory misses for each request will depend on code path, as will capacity misses due to a request's working set size. However, cache misses due to sharing between requests are not captured in a request-mix model. Such misses are not present in the single processor tests in Figure 4.

Table 3 evaluates request-mix models under platform configurations that allow for shared misses. The first three columns report low normalized error when resources are adequately provisioned (below 0.13 for all applications), as they would be in production environments. However under maximum throughput conditions, accuracy suffers. Specifically, we increased the volume of the tested request mixes by a factor of 3. Approximately 50% of the test request mixes achieved 100% processor utilization. Normalized error increased for the L1 miss metric to 0.22–0.34. These results are consistent with past work [29] that attributed shared misses in Internet services to contention for software resources. In the realistic, adequately provisioned environments that we target, such contention is rare. We conclude that request-mix models are most appropriate in realistic environments when resources are not saturated.

Request-mix models can be calibrated with time-stamped observations of the targeted processor metric and logs of request arrivals, both of which are commonly available in practice. Past work [40] demonstrated that unrealistic stationary workloads are not sufficient for calibration. Request-mix models can require many observations before converging to good parameters. For the real and benchmark applications that we have seen, request mix models based on ten hours of log files are typically sufficient.

	2-way SMP (L1 miss)	Dual-core (L2 miss)	Hyperthreading (L1 miss)	Max tput (L1 miss)
RUBiS	0.044	0.030	0.045	0.349
Stock	0.060	0.077	0.096	0.276
TPCW	0.035	0.084	0.121	0.223

Table 3: Median normalized residual error of a request-mix model under different environments. Evaluation spans 583 request mixes from a nonstationary trace. The target architectural metric is shown in parenthesis. All tests were performed on Pentium D processors. Hyperthreading was enabled in the system BIOS. 2-way SMP and dual-cores were enabled/disabled by the operating system scheduler. The “max tput” test was performed on a configuration with 2 processors and 4 cores enabled with hyperthreading on.

4 Cross-Platform Performance Predictions

Section 3 described trait models, which are parsimonious characterizations of only one aspect of a complex system. In this section, we will show that trait models can be composed to predict application-level performance for the whole system. Our novel composition is formed from expert knowledge about the structure of Internet services. In particular, we note that instruction and memory-access latencies are key components of the processing time of individual requests, and that Internet services fit many of the assumptions of a queuing system. Our model predicts application-level performance across workload changes and several platform parameters including: processor speed, number of processors (*e.g.*, on-chip cores), cache size, cache latencies, and instruction latencies. Further, our model can be calibrated from the logs described in Table 1. Source code access, instrumentation, and controlled experiments are not required. We attribute the low prediction error of our model to accurate trait models (described in Section 3) and a principled composition based on the structure of Internet services.

The remainder of this section describes the composition of our cross-platform model, then presents the test platforms that we use to validate our model. Finally, we present results by comparing against alternative modeling techniques. In Section 5, we complete the challenge of turning 15-cent production data into a dollar by using our model to guide management decisions like platform selection and load balancing.

4.1 Composition of Traits

The amount of time spent processing an end-user request, called service time, depends on the instructions and memory accesses necessary to complete the request.

Average CPU service time can be expressed as

$$s = \frac{I \times (\text{CPI} + (H_1 C_1 + H_2 C_2 + M_2 C_{\text{mem}}))}{\text{CPU speed} \times \text{number of requests}}$$

where I is the aggregate number of instructions required by a request mix, CPI is the average number of cycles per instruction (not including memory access delays), H_k is the percentage of hits in the L_k cache per instruction, M_k is the percentage of misses in the L_k cache per instruction, and C_k is the typical cost in cycles of accesses to the L_k cache [22]. CPI , CPU speed , and C_k are typically released in processor spec sheets [6]. Recent work [16] that more accurately approximates CPI and C_k could transparently improve the prediction accuracy of our model.

This model is the basis for our cross-platform performance prediction. Subsequent subsections will extend this base to handle changes in application-level workload and cache size parameters, and to predict the application-level performance.

4.1.1 Request Mix Adjustments

In Section 3, we observed that both instruction counts and cache misses, at both L1 and L2, are well modeled as a linear combination of request type frequencies:

$$I = \sum_{\text{types } j} \alpha_{Ij} N_j \quad \text{and} \quad \#M_k = \sum_{\text{types } j} \alpha_{M_k j} N_j$$

where I is the number of instructions for a given volume and mix of requests, N_j is the volume of requests of type j , and $\#M_k$ is the number of misses at cache level k . The intuition behind these models is straightforward: α_{Ij} , for example, represents the typical number of instructions required to serve a request of type j . We apply ordinary least squares regression to a 10-hour trace of nonstationary request mixes to calibrate values for the α parameter. After calibration, the acquired $\hat{\alpha}$ parameters can be used to predict performance under future request mixes. Specifically, we can predict both instruction count and aggregate cache misses for an unseen workload represented by a new vector N' of request type frequencies.

4.1.2 Cache Size Adjustments

Given L1 and L2 cache miss rates observed on the current hardware platform, we predict miss rates for the cache sizes on a new hardware platform using the power-law cache model: $M_k = e^A S_k^B$ where S_k is the size of the level- k cache.

We calibrate the power law under the most strenuous test possible: using cache-miss observations from only an L1 and L2 cache. This is the constraint under which many consultants must work: they can measure an application running in production on only a single hardware platform. Theoretically, the stable calibration of

power law models desires observations of cache misses on 5 cache sizes [19]. Calibration from only two observations skews the model's predictions for smaller cache sizes [33]. However in terms of service time prediction, the penalty of such inaccuracies—L1 latency—is low.

4.1.3 Additional Service Time Adjustments

Most modern processors support manual and/or dynamic frequency adjustments. Administrators can manually throttle CPU frequencies to change the operation of the processor under idle and busy periods. Such manual policies override the *CPU speed* parameter in processor spec sheets. However, power-saving approaches in which the frequency is automatically adjusted only during idle periods are not considered in our model. Such dynamic techniques should not affect performance during the busy times in which the system is processing end-user requests.

We consider multi-processors as one large virtual processor. Specifically, the *CPU speed* parameter is the sum of cycles per second across all available processors. We do not distinguish between the physical implementations of logical processors seen by the OS (*e.g.*, SMT, multi-core, or SMP). We note however that our model accuracy could be improved by distinguishing between cores and hyperthreads.

4.1.4 Predicting Response Times

Service time is not the only component of a request's total response time; often the request must wait for resources to become available. This aspect of response time, called queuing delay, increases non-linearly as the demand for resources increases. Queuing models [27] attempt to characterize queuing delay and response time as a function of service time, request arrival rate, and the availability of system resources. Our past work presented a queuing model that achieves accurate response time prediction on *real* Internet services [40]. That particular model has two key advantages: 1) it considers the weighted impact of request mix on service times and 2) it can be easily calibrated in the production environments that we target. It is a model of aggregate response time y for a given request mix and is shown below:

$$y = \sum_{j=1}^n s_j N_j + \sum_r \left(\frac{1}{\lambda} \cdot \frac{U_r^2}{1 - U_r} \right) \cdot \sum_{j=1}^n N_j$$

where λ and U_r respectively denote the aggregate count of requests in the given mix (*i.e.*, arrival rate) and the average utilization of resource r , respectively. Utilization is the product of average service time and arrival rate. The first term reflects the contribution of service times to aggregate response time, and the second considers queuing

delays. For average response time, divide y by λ . The parameter s_j captures average service time for type j and can be estimated via regression procedures using observations of request response times and resource utilizations [40]. Note that y reflects aggregate response time for the whole system; s_j includes delays caused by other resources—not just processing time at the business-logic tier. Our service time predictions target the portion of s_j attributed to processing at the business-logic tier only.

4.2 Evaluation Setup

We empirically evaluate our service and response time predictions for RUBiS, StockOnline, and TPC-W. First, we compare against alternative methods commonly used in practice. Such comparisons suggest that our model is immediately applicable for use in real world problems. Then we compare against a model recently proposed in the research literature [25]. This comparison suggests that our principled modeling methodology provides some benefits over state-of-the-art models.

4.2.1 Test Platforms

We experiment with 4 server machines that allow for a total of 11 different platform configurations. The various servers are listed below:

PIII Dual-processor PIII Coppermine with 1100 MHz clock rate, 32 KB L1 cache, and 128 KB L2 cache.

PRES Dual-processor P4 Prescott with 2.2 GHz clock rate, 16 KB L1 cache, and 512 KB L2 cache.

PD4 Four-processor dual-core Pentium D with 3.4 GHz clock rate, 32 KB L1 cache, 4 MB L2 cache, and 16 MB L3 cache.

XEON Dual-processor dual-core Pentium D Xeon that supports hyperthreading. The processor runs at 2.8 GHz and has a 32 KB L1 cache and 2 MB L2 cache.

We used a combination of BIOS features and OS scheduling mechanisms to selectively enable/disable hyperthreading, multiple cores, and multiple processors on the XEON machine, for a total of eight configurations. We describe configurations of the XEON using notation of the form “#H/#C/#P.” For example, 1H/2C/1P means hyperthreading disabled, multiple cores enabled, and multiple processors disabled. Except where otherwise noted, we calibrate our models using log files from half of a 10-hour trace on the PIII machine. Our log files contain aggregate response times, request mix information, instruction count, aggregate L1 cache misses, and aggregate L2 cache misses. The trace contains over 500 mixes cumulatively. Request mixes from the remaining half of the trace were used to evaluate the normal-

ized residual error $\frac{|\text{predicted}-\text{actual}|}{\text{actual}}$ on the remaining ten platforms/configurations. Most mixes in the second half of the trace constitute extrapolation from the calibration data set; specifically, mixes in the second half lie outside the convex hull defined by the first half.

4.2.2 Alternative Models Used In Practice

Because complex models are hard to calibrate in data-constrained production environments, the decision support tools used in practice have historically preferred simplicity to accuracy. Commonly used tools involve simple reasoning about the linear consequences of platform parameters on CPU utilization and service times.

Processor Cycle Adjustments Processor upgrades usually mean more clock cycles per second. Although the rate of increase in clock speeds of individual processor cores has recently slowed, the number of cores per system is increasing. Highly-concurrent multi-threaded software such as business-logic servers processing large numbers of simultaneous requests can exploit the increasing number of available cycles, so the net effect is a transparent performance improvement. A common approach to predicting the performance impact of a hardware upgrade is simply to assume that CPU service times will decrease in proportion to the increase in clock speed. For service time prediction, this implies

$$st_{\text{new}} = st_{\text{orig}} \times \frac{\text{cycles}_{\text{orig}}}{\text{cycles}_{\text{new}}} \quad (3)$$

However, this approach does not account for differences in cache sizes on the old and new CPUs. Section 3 has shown that changes to the cache size have non-linear effects on the number of cache misses for business-logic servers, which in turn affects CPU service times and CPU utilization. Furthermore, cache miss latencies differ across hardware platforms, and these differences too may affect CPU service times.

Benchmark-Based Adjustments A more sophisticated approach used widely in practice is to predict performance using ratios of standard application benchmark scores. Specifically, these approaches measure the service times and maximum throughput for a target application on both the new and original platform. This application's performance is expected to be representative of a larger class of applications. This model is shown below:

$$st_{\text{new}} = st_{\text{orig}} \times \frac{\text{TPC score}_{\text{orig}}}{\text{TPC score}_{\text{new}}} \quad (4)$$

This approach improves on the processor cycle adjustments by considering the effect of caching and other

architectural factors. However, modern business-logic servers can vary significantly from standard benchmark applications. Indeed, standard benchmarks differ substantially *from one another* in terms of CPI and miss frequencies [39]!

Workload Adjustments Workload fluctuations are ubiquitous in Internet services, and we frequently wish to predict the effects of workload changes on system resources. One typical approach is to model CPU utilization as a linear function of request volume.

$$\text{utilization}_{\text{new}} = \text{utilization}_{\text{orig}} \times \frac{\text{request rate}_{\text{new}}}{\text{request rate}_{\text{orig}}} \quad (5)$$

The problem with this approach is that while it accounts for changes in the aggregate volume of requests, it ignores changes in the *mix of request types*. We call such a model a scalar performance model because it treats workload as a scalar request rate rather than a vector of type-specific rates. Nonstationarity clearly poses a serious problem for scalar models. For example, if request volume doubles the model will predict twice as much CPU utilization. However if the increase in volume was due entirely to a lightweight request type, actual utilization may increase only slightly.

4.3 Results

4.3.1 Accuracy of Service Time Predictions

We first consider the problem of predicting average per-request CPU service times of an application on a new hardware platform given observations of the application running on our PIII platform. We compare our method with three alternatives: 1) the naïve cycle-based adjustment of Equation 3 with linear workload adjustments of Equation 5, 2) a variant in which we first predict service time as a function of request mix using a linear weighted model and then apply the cycle-based adjustment, and 3) the benchmark-based method of Equation 4 with request mix workload adjustments.

Table 4 shows the prediction error for RUBiS. Our method has less error than the benchmark-based method for all target platforms. In all cases except one, our method has lower error than its three competitors. Most often, our model has less than one third the error of competing approaches. The results for StockOnline and TPC-W are similar. Table 5 shows that our predictions are always the most accurate for the StockOnline application. Our results for TPC-W (Table 6) show consistently low error for our model (always below 0.17).

Target Processor	Cycle pred		Bench-	Our
	scalar	req-mix	mark	Method
PRES	0.48	0.35	0.30	0.06
PD4	0.24	0.17	0.32	0.07
XEON 1H/1C/1P	0.63	0.49	0.27	0.10
XEON 1H/1C/2P	0.43	0.28	0.24	0.08
XEON 1H/2C/1P	0.39	0.30	0.26	0.02
XEON 1H/2C/2P	0.36	0.27	0.57	0.03
XEON 2H/1C/1P	0.42	0.28	0.24	0.02
XEON 2H/1C/2P	0.12	0.05	0.29	0.24
XEON 2H/2C/1P	0.35	0.22	0.32	0.02
XEON 2H/2C/2P	0.18	0.13	0.38	0.09

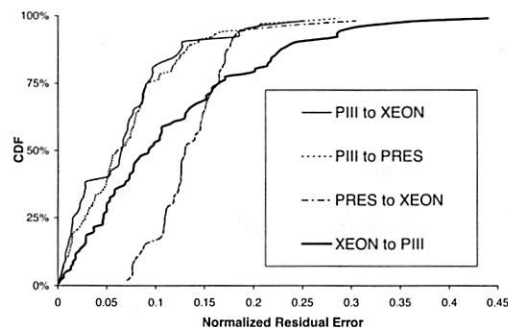
Table 4: Median normalized residual error of service time predictions for RUBiS. Models are calibrated on our PIII platform from half of a 10-hour nonstationary trace. The numbers presented are the median error when the models are used to predict the second half of the nonstationary trace on the targeted platforms. The median across platforms for our method is 0.07.

Target Processor	Cycle pred		Bench-	Our
	scalar	req-mix	mark	Method
PRES	0.35	0.23	0.56	0.18
PD4	0.38	0.28	0.42	0.12
XEON 1H/1C/1P	0.55	0.41	0.21	0.10
XEON 1H/1C/2P	0.48	0.38	0.34	0.12
XEON 1H/2C/1P	0.44	0.39	0.35	0.09
XEON 1H/2C/2P	0.36	0.26	0.71	0.12
XEON 2H/1C/1P	0.48	0.38	0.34	0.17
XEON 2H/1C/2P	0.22	0.16	0.52	0.14
XEON 2H/2C/1P	0.45	0.37	0.47	0.15
XEON 2H/2C/2P	0.36	0.29	0.39	0.02

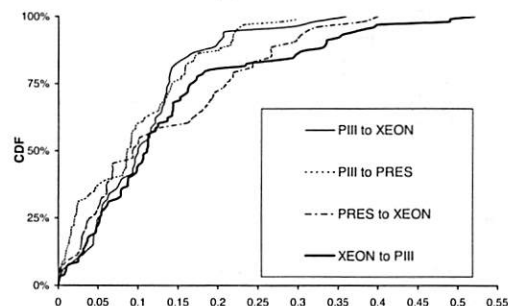
Table 5: Median normalized residual error of service time predictions for StockOnline. The median across platforms for our method is 0.12.

Target Processor	Cycle pred		Bench-	Our
	scalar	req-mix	mark	Method
PRES	0.48	0.35	—	0.11
PD4	0.28	0.24	—	0.09
XEON 1H/1C/1P	0.38	0.32	—	0.13
XEON 1H/1C/2P	0.23	0.16	—	0.13
XEON 1H/2C/1P	0.22	0.16	—	0.12
XEON 1H/2C/2P	0.27	0.22	—	0.17
XEON 2H/1C/1P	0.28	0.23	—	0.06
XEON 2H/1C/2P	0.24	0.20	—	0.11
XEON 2H/2C/1P	0.20	0.15	—	0.16
XEON 2H/2C/2P	0.18	0.21	—	0.14

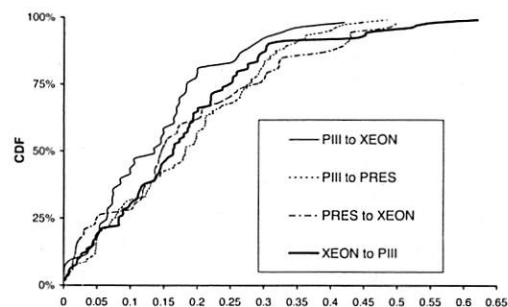
Table 6: Median normalized residual error of service time predictions for TPC-W. Note, the benchmark method is not applicable for TPC-W, since observations on the new platform are required for calibration. The median across platforms for our method is 0.13.



(a) RUBiS



(b) StockOnline



(c) TPCW

Figure 5: Cumulative distributions of normalized residual error for response time predictions across nonstationary request mixes. Mean response times on the PIII, P4, and XEON were 256, 106, and 93 ms respectively. XEON shortens XEON 2H/2T/2P according to our naming conventions.

4.3.2 Accuracy of Response Time Predictions

Figure 5 shows the prediction accuracy of our queuing model. For predictions from PIII to XEON, our model converts median service time error of 0.09, 0.02, and 0.14 into response time predictions with median error of 0.06, 0.09, and 0.13, for RUBiS, TPC-W, and StockOnline respectively. Even though response time is a more complex metric, our prediction error remains low.

Figure 5 also demonstrates the robustness of our method. The 85th percentile of our prediction error for RUBiS is always below 0.21 no matter which platforms we use for calibration and validation. For StockOnline and TPC-W, the 85th percentile prediction error is below 0.29 and 0.30, respectively.

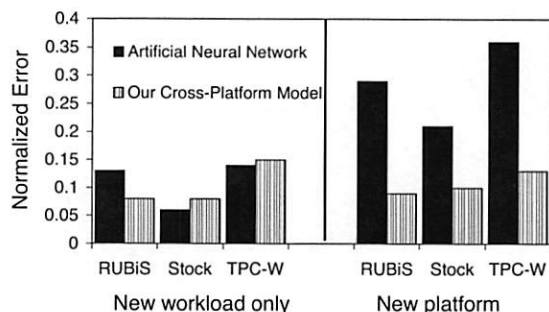


Figure 6: Comparison of an artificial neural network to our cross-platform model. The median normalized error for predictions of average response time are reported. We trained the ANN on several subsets of the training data. The reported values are from the training subset that yielded the lowest prediction error for each application.

4.3.3 Comparison to Neural Net Model

Ipek *et al.* used an artificial neural network to predict performance across changes in platform parameters [25]. Artificial neural networks can be automatically calibrated without knowledge of the form of the functional relationship between input parameters and output variables. Also, they operate with categorical and continuous data. However, the limited amount of available data for cross-platform management for real-world Internet services presents a challenge for neural networks. Neural networks require many observations of the input parameters and output variables in order to learn the underlying structure of the system. In contrast, our composition of trait models is based on our knowledge of the underlying structure of Internet services.

We implemented an artificial neural network (ANN) as described in [25]. We used 16 hidden states, a learning rate 0.0001, a momentum value of 0.5, and we initialized the weights uniformly. The output variable for the ANN was the average response time. The training set for the ANN consisted of observations on the PIII and PRES platforms. The validation set consisted of observations under new transaction mixes on the PRES and XEON platforms.

Figure 6 shows the median prediction error of the ANN compared to our model on the validation set. Our model has comparable accuracy, within 0.02, under situations in which the ANN predicts only the effects of workload changes. However, the ANN has up to 3X the error of our model when predicting response time on the unseen XEON platform. Observations on two platforms are not enough for the ANN to learn the relationship between platform parameters and response time. These results suggest that our methodology, a composition of trait models, may be better suited for cross-platform management in data-constrained production environments.

5 Enhanced System Management

In this section, we use our model to improve cross-platform management decisions for Internet services. In general, such management decisions can have significant consequences on the bottom line for service providers, which completes our metaphor of creating a dollar from 15 cents. We explore two specific management problems often encountered in real-world production environments.

- *Platform Selection* When building or augmenting a server cluster, service providers wish to select platforms that will maximize performance relative to a cost. We look at the problem of choosing the hardware platform that yields maximum response-time-bounded throughput per watt. This problem is challenging because architectural features and configuration options that enhance performance also increase power consumption. Of course, the problem could also be solved by testing the application of interest on each target processor configuration, but such exhaustive testing is typically too expensive and time consuming in real-world scenarios.
- *Platform-Aware Load Balancing for Heterogeneous Servers* Service providers wish to extract maximum performance from their server infrastructure. We show that naïvely distributing arriving requests across all machines in a server cluster may yield sub-optimal performance for certain request types. Specifically, certain types of requests execute most efficiently only under certain platform configurations in the server cluster. Our cross-platform performance predictions can identify the best platform for each request type.

5.1 Platform Selection

Our metric for platform selection is throughput per watt. We leverage our performance model of application-level response time to predict the maximum request arrival rate that does not violate a bound on aggregate response time. Given an expected request mix, we iteratively query our model with increased aggregate arrival rates. The response-time-bounded throughput is the maximum arrival rate that does not exceed the response time bound. The other half of our metric for platform selection is power consumption, which we acquire from processor spec sheets [6]. Admittedly, processor specs are not the most accurate source for power consumption data [17, 21], but they will suffice for the purpose of demonstrating our model's ability to guide management decisions.

For this test, we compare our model against the other cross-platform prediction methods commonly used in practice. The competing models are not fundamentally

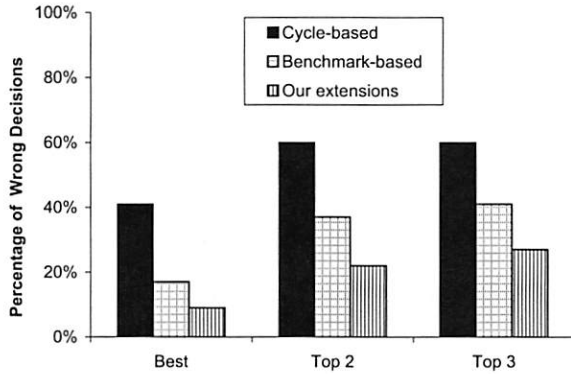


Figure 7: Probability of incorrect decisions for model-driven platform selection.

intended to model response time, so we apply our queuing model to each. Specifically, the difference in results in this section come from the different methods of service time prediction presented in Section 4. We calibrate performance models on the PIII platform, and show results for the RUBiS auction service. Table 7 compares the measured throughput per watt to the predictions of each method. The absolute value of the difference between the actual ranking and the predicted ranking (i.e., $|pred - act|$), is never greater than 1 with our method. Comparatively, the cycle-based and benchmark approach have mis-rankings of 5 and 3 respectively. In practice, mis-rankings could cause service providers to purchase an inefficient platform. One way to measure the cost of such mis-predictions is to measure the net loss in throughput per watt between the mis-predicted platform and the properly ranked platform, i.e., $\frac{tpw^{predicted\ rank\ k}}{tpw^{actual\ rank\ k}} \times 100\%$. The maximum loss is only 5% for our method, but is 45% and 12% for the cycle-based and benchmark methods, respectively.

Often in practice, platform selections are made from a few properly priced platforms. We divided our platform into subsets of five, and then predicted the best platform in the subset. We report how often each performance prediction method does *NOT* correctly identify the best, the top two, and the top three platforms. Figure 7 shows that our method selects the best processor configuration for 230 of 252 (91%) combinations. More importantly, alternative methods are 2X and 4X more likely to make costly wrong decisions. For the problems of correctly identifying the top two and top three platforms, our approach yields the correct decision for all but 25% of platform subsets whereas our competitors are wrong more than 41% and 60% of the time.

Processor	Rankings			
	actual	our method	cycle-based	bench-mark
PD4	1	1	1	1
XEON 2H/2C/2P	2	2	6	2
XEON 2H/2C/1P	3	4	3	5
XEON 2H/1C/1P	4	3	4	3
XEON 2H/1C/2P	5	5	5	6
XEON 1H/1C/1P	6	7	7	7
PRES	7	6	2	4
XEON 1H/2C/1P	8	8	8	8
XEON 1H/1C/2P	9	9	9	9
XEON 1H/2C/2P	10	10	10	10

Table 7: Platform rankings of the response-time-bounded throughput per watt on RUBiS. Response time bound was 150ms. Power consumption data was taken from [6]. Actual response-time-bounded throughput was measured as a baseline.

5.2 Platform-Aware Load Balancing

In this section, we first observe that *the best processor for one request type may not be the best processor for another*. Second, we show that the techniques described in Section 4 enable black-box ranking of processors for each request type. Our techniques complement advanced type-based load distribution techniques, like locality-aware request distribution, in heterogeneous clusters and do not require intrusive instrumentation or benchmarking. A mapping of request type to machine can be made with the limited data available to consultants.

Table 8 shows the expected and actual response time for four request types that contribute significantly to the overall response time of RUBiS. These types were chosen because they are representative of different demands for processor resources in RUBiS. The first type represents requests for static content. Since XEON and PRES have similar clock rates there is not much difference between their processing power on this request type. *Search Items by Region* has a large working set and benefits from fewer L2 cache misses on XEON. *View Item* has a small working set size that seems to fit mostly within the larger L1 cache of XEON. *Lookup user information*, however, has unique cache behavior. Its working set is medium-sized consisting of user information, user bid histories, and user items currently for sale. This request type seems to benefit more from the lower-latency 512KB cache of PRES, than it does from the larger 32KB L1 cache of Xeon.

Table 8 also shows our ability to predict the best machine on a per-request-type basis. We achieve this by us-

	Response Time Per-type (ms)			
	PRES		Dual-core Xeon	
	actual	predict	actual	predict
Browse.html	53	49	48	47
Search Items by Reg.	423	411	314	354
View Item	102	110	89	92
Lookup User Info	80	75	132	109

Table 8: Expected response time of each request-type when issued in isolation. Predictions are based on log file observations from PIII. The dual-core Xeon is a single processor. Request mix consisted of only requests of the tested type at a rate of ten requests per second.

ing the techniques described in Section 4. In particular, we calibrate the parameters of our business-logic traits using only lightweight passive observations of a system serving a realistic nonstationary workload. Then, we predict performance under a workload mix that consists of only one type of request. This is done for each machine and request type in the system.

6 Related Work

Our contributions in this paper span model-driven system management, workload characterization, and performance modeling. The general literature on these topics is vast; this section briefly reviews certain related works.

6.1 Model-Driven Management

Networked services are now too complex for ad-hoc, human-centric management. Performance predictions, the by-product of performance models, offer a convenient abstraction for principled, human-free management. Our previous work [41] presented a whole-system performance model capable of guiding the placement and replication of interacting software components (*e.g.*, web server, business-logic components, and database) across a cluster. Shivam *et al.* [34, 37] model scientific applications on a networked utility. Their model can be used to guide the placement of compute and I/O-bound tasks and the order in which workflows execute. Doyle *et al.* provide a detailed model of file system caching for standard web servers that is used to achieve resource management objectives such as high service quality and performance isolation [14]. Magpie [9] models the control flow of requests through distributed server systems, automatically clustering requests with similar behavior and detecting anomalous requests. Aguilera *et al.* [7] perform bottleneck analysis under conditions typical of real-world Internet services. Thereska and Ganger [42] investigate real-world causes for inaccuracies in storage system models and study their effect on management

policies. Finally, our most recent work proposed a model of application-level performance that could predict the effects of combining two or more applications onto the same machine (*i.e.*, consolidation).

The contribution of this work is a practical method for performance prediction across platform and workload parameters that can be used to guide cross-platform decisions for real-world Internet services.

6.2 Workload Characterization

Internet services have certain innate resource demands that span hardware platforms, underlying systems software, and even the end-user input supplied to them. Previous work [32, 36] advocated separating the characterization of application-specific demands from the characterization of underlying systems and high-level inputs. Our trait models continue in this tradition by providing parsimonious and easy-to-calibrate descriptions of an application's demand for hardware resources. We believe trait models exist and can be derived for other types of applications, such as databases and file systems.

Characterizations of application resource demand that have functional forms similar to our trait models have been observed in past research. Saavedra & Smith model the execution time of scientific FORTRAN programs as a weighted linear combination of "abstract operations" such as arithmetic and trigonometric operations [35]. Our request-mix models characterize much higher level business-logic operations in a very different class of applications. Another difference is that Saavedra & Smith calibrate their models using direct measurements obtained through invasive application instrumentation whereas we analyze lightweight passive observations to calibrate our models.

Chow considers the optimal design of memory hierarchies under the *assumption* of a power-law relationship between cache size and miss rates [13]. Smith presents very limited empirical evidence, based on a single internal Amdahl benchmark, that appears to be roughly consistent with Chow's assumption [38]. Thiebaut and Hartstein *et al.* explore a special case of Chow's assumption from a theoretical standpoint [20, 43]. This prior work is primarily concerned with the *design* of cache hierarchies and typically employs traces of memory accesses. We compose a power-law model with request-mix models and queuing models to *predict* the impact on response times of both workload changes and architectural changes. Furthermore we employ only passive observations of a running application to estimate power-law parameters.

The growing commercial importance of Java-based middleware and applications has attracted considerable attention in the architecture community. Recent stud-

ies investigate interactions between Java benchmarks and architectural features including branch prediction, hyperthreading, and the CPU cache hierarchy [11, 24, 29]. One important difference between these investigations and our work is that they focus on throughput whereas we emphasize application-level response times. Also, our work incorporates expert knowledge about Internet services via a novel composition of trait models. We demonstrated, in Section 4.3.3, that our composition improves cross-platform performance prediction when only limited production data is available.

6.3 Cross-Platform Performance Models

Chihai & Gross report that simple analytic memory models accurately predict the execution times of scientific codes across hardware architectures [12]. We predict response times in business-logic servers across platforms, and we too find that succinct models suffice for our purposes. More sophisticated approaches typically fall into one of two categories: Knowledge-free machine learning techniques [23, 25, 30] and detailed models based on deep knowledge of processor design [16, 28, 32].

Detailed knowledge-intensive models typically require more extensive calibration data than is available to consultants in the practical scenarios that motivate our work, *e.g.*, data available only through simulated program execution. Knowledge-free machine learning methods, on the other hand, typically offer only limited insight into application performance: The structure and parameters of automatically-induced models are often difficult to explain in terms that are meaningful to a human performance analyst or useful in an IT management automation problem such as the type-aware load distribution problem of section 5.2. Furthermore the accuracy of knowledge-free data-mining approaches to cross-platform performance prediction has been a controversial subject: See, *e.g.*, Ein-Dor & Feldmesser for sweeping claims of accuracy and generality [15] and Fullerton for a failure to reproduce these results [18].

Our contribution is a principled composition of concise and justifiable models that are easy to calibrate in practice, accurate, and applicable to online management as well as offline platform selection.

7 Conclusion

This paper describes a model-driven approach for cross-platform management of real-world Internet services. We have shown that by composing *trait models*—parsimonious, easy-to-calibrate characterizations of one aspect of a complex system—we can achieve accurate cross-platform performance predictions. Our trait mod-

els themselves are typically accurate to within 10% and our application-level performance predictions are typically accurate to within 15%. Our approach relies only on lightweight passive observations of running production systems for model calibration; source code access, invasive instrumentation, and controlled benchmarking are not required. Applied to the problem of selecting a platform that offers maximal throughput per watt, our approach correctly identifies the best platform 91% of the time whereas alternative approaches choose a sub-optimal platform 2x–4x more frequently. Finally, we have shown that our model can improve load balancing in a heterogeneous server cluster by assigning request types to the most suitable platforms.

8 Acknowledgments

More people assisted in this work than we can acknowledge in this section. Keir Fraser, our shepherd, helped us polish the camera-ready version. The anonymous reviewers gave rigorous, insightful, and encouraging feedback, which improved the camera-ready version. Reviews from our friends Kim Keeton, Xiaoyun Zhu, Zhikui Wang, Eric Anderson, Ricardo Bianchini, Nidhi Aggarwal, and Timothy Wood helped us nail down the contributions. Jaap Suermondt helped formalize an evaluation methodology for the decision problems in Section 5. Eric Wu (HP) and Jim Roche (Univ. of Rochester) maintained the clusters used in our experiments. Part of this work was supported by the National Science Foundation grants CNS-0615045 and CCF-0621472.

References

- [1] Apache software foundation. <http://www.apache.org>.
- [2] Oprofile: A system profiler for linux. <http://oprofile.sourceforge.net/>.
- [3] Rice university bidding system. <http://rubis.objectweb.org/>.
- [4] Stock-online. <http://objectweb.org/stockonline>.
- [5] <http://www.cs.rochester.edu/u/stewart/models.html>.
- [6] x86 technical information. <http://www.sandpile.org>.
- [7] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.
- [8] A. Andrzejak, M. Arlitt, and J. A. Rolia. Bounding the resource savings of utility computing models. Technical report, HP Labs, Dec. 2002.

- [9] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modeling. In *OSDI*, Dec. 2004.
- [10] G. Box and N. Draper. Empirical model-building and response surfaces. Wiley, 1987.
- [11] H. Cain, R. Rajwar, M. Marden, and M. Liphasti. An architectural evaluation of java tpc-w. In *HPCA*, Dec. 2001.
- [12] I. Chihaiia and T. Gross. Effectiveness of simple memory models for performance prediction. In *ISPASS*, Mar. 2004.
- [13] C. Chow. On optimization of storage hierarchy. In *IBM J. Res. Dev.*, May 1974.
- [14] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-based resource provisioning in a web service utility. In *USENIX Symp. on Internet Tech. & Sys.*, Mar. 2003.
- [15] P. Ein-Dor and J. Feldmesser. Attributes of the performance of central processing units: A relative performance prediction model. *CACM*, 30(4):308–317, Apr. 1987.
- [16] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, Oct. 2006.
- [17] X. Fan, W. Weber, and L. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA*, June 2007.
- [18] G. D. Fullerton. An evaluation of the gains achieved by using high-speed memory in support of the system processing unit. *CACM*, 32(9):1121–1129, 1989.
- [19] M. Goldstein, S. Morris, and G. Yen. Problems with fitting to the power-law distribution. In *The European Physical Journal B - Condensed Matter and Complex Systems*, June 2004.
- [20] A. Hartstein, V. Srinivasan, T. Puzak, and P. Emma. Cache miss behavior: is it sqrt(2)? In *Conference on Computing Frontiers*, May 2006.
- [21] T. Heath, A. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and freon: Temperature emulation and management for server systems. In *ASPLOS*, Oct. 2006.
- [22] J. Hennessey and D. Patterson. Computer architecture: A quantitative approach, fourth edition. In *The Morgan Kaufmann Series*, 2007.
- [23] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. John, and K. Bosschere. Performance prediction based on inherent program similarity. In *Int'l Conf. on Parallel Architectures & Compilation Techniques*, Sept. 2006.
- [24] W. Huang, J. Liu, Z. Zhang, and M. Chang. Performance characterization of Java applications on SMT processors. In *ISPASS*, Mar. 2005.
- [25] E. Ipek, S. McKee, B. Supinski, M. Schultz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, Oct. 2006.
- [26] R. Isaacs and P. Barham. Performance analysis in loosely-coupled distributed systems. In *Cabernet Radicals Workshop*, Feb. 2002.
- [27] R. Jain. The art of computer systems performance analysis. In Wiley, 1991.
- [28] T. Karkhanis and J. Smith. A first-order superscalar processor model. In *ISCA*, June 2004.
- [29] M. Karlsson, K. Moore, E. Hagersten, and D. Wood. Memory system behavior of java-based middleware. In *HPCA*, Feb. 2003.
- [30] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS*, Oct. 2006.
- [31] X. Li, Z. Li, P. Zhou, Y. Zhou, S. Adve, and S. Kumar. Performance-directed energy management for main memory and disks. In *ASPLOS*, Oct. 2004.
- [32] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS*, June 2004.
- [33] M. Newman. Power laws pareto distributions and zipf's law. In *Contemporary Physics*, 2005.
- [34] S. B. P. Shivam and J. Chase. Active and accelerated learning of cost models for optimizing scientific applications. In *VLDB*, Sept. 2006.
- [35] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comp. Sys.*, 14(4):344–384, Nov. 1996.
- [36] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The case for application-specific benchmarking. June 1999.
- [37] P. Shivam, A. Iamnitchi, A. Yumerefendi, and J. Chase. Model-driven placement of compute tasks in a networked utility. In *ACM International Conference on Autonomic Computing*, June 2005.
- [38] A. Smith. Cache memories. In *ACM Computing Surveys*.
- [39] R. Stets, L. Barroso, and K. Gharachorloo. A detailed comparison of two transaction processing workloads. In *IEEE Workshop on Workload Characterization*, Nov. 2002.
- [40] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys*, Mar. 2007.
- [41] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI*, May 2005.
- [42] E. Thereska and G. Ganger. Ironmodel: Robust performance models in the wild. In *SIGMETRICS*, June 2008.
- [43] D. Thiebaut. On the fractal dimension of computer programs. In *IEEE Trans. Comput.*, July 1989.
- [44] B. Urgoankar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS*, June 2005.

Measurement and Analysis of Large-Scale Network File System Workloads

Andrew W. Leung* Shankar Pasupathy† Garth Goodson† Ethan L. Miller*

**University of California, Santa Cruz
{aleung, elm}@cs.ucsc.edu*

*†NetApp Inc.
{shankarp, goodson}@netapp.com*

Abstract

In this paper we present the analysis of two large-scale network file system workloads. We measured CIFS traffic for two enterprise-class file servers deployed in the NetApp data center for a three month period. One file server was used by marketing, sales, and finance departments and the other by the engineering department. Together these systems represent over 22 TB of storage used by over 1500 employees, making this the first ever large-scale study of the CIFS protocol.

We analyzed how our network file system workloads compared to those of previous file system trace studies and took an in-depth look at access, usage, and sharing patterns. We found that our workloads were quite different from those previously studied; for example, our analysis found increased read-write file access patterns, decreased read-write ratios, more random file access, and longer file lifetimes. In addition, we found a number of interesting properties regarding file sharing, file re-use, and the access patterns of file types and users, showing that modern file system workload has changed in the past 5–10 years. This change in workload characteristics has implications on the future design of network file systems, which we describe in the paper.

1 Introduction

Network file systems are playing an increasingly important role in today's data storage. The motivation to centralize data behind network file systems has been driven by the desire to lower management costs, the need to reliably access growing amounts of data from multiple locations, and is made possible by improvements in processing and network power. The design of these systems is usually guided by an understanding of file system workloads and user behavior [12, 19, 25], which is often obtained by measuring and analyzing file system traces.

While a number of trace-based file system studies have been conducted in the past [3, 5, 21, 24, 29], there are

several factors indicating that further study is necessary to aid future network file system designs. First, no major study has been conducted of CIFS (Common Internet File System) [13], the network file transfer protocol used by Windows. Second, the last major trace study [5] analyzed traces from 2001, over half a decade ago; significant changes in the architecture and use of network storage since then have resulted in changes in workload. Third, no published study has ever analyzed large-scale enterprise file system workloads, focusing instead on research-type workloads, such as those seen in the university settings often available to systems researchers.

In this paper, we examine two real-world, large-scale enterprise network file system workloads, collected over three months from two network file servers deployed in NetApp's data center. One server hosts data for the marketing, sales, and finance departments, and the other hosts data for engineering departments. Combined, these systems contain over 22 TB of actively used storage and are used by over 1500 employees. The analysis of our trace data focused on: (1) changes in file access patterns and lifetimes since previous studies, (2) properties of file I/O and file sharing, and (3) the relationship between file type and client access patterns.

Our analysis found important changes in several aspects of file system workloads. For example, we found that read-write file access patterns, which are highly random, are much more common relative to read-only and write-only access patterns as compared to past studies. We also found our workloads to be more write-oriented than those previously studied, with only about twice as many bytes read as written. Both of these findings challenge traditionally-held assumptions about access patterns and sequentiality. A summary of our key observations can be found in Table 1.

In all, our contributions include:

- 1) The first published study of CIFS workloads.
- 2) A comparison with past file system studies.
- 3) A new study of file access, I/O and sharing patterns.

Compared to Previous Studies
1. Both of our workloads are more write-oriented. Read to write byte ratios have significantly decreased. 2. Read-write access patterns have increased 30-fold relative to read-only and write-only access patterns. 3. Most bytes are transferred in longer sequential runs. These runs are an order of magnitude larger. 4. Most bytes transferred are from larger files. File sizes are up to an order of magnitude larger. 5. Files live an order of magnitude longer. Fewer than 50% are deleted within a day of creation.
New Observations
6. Files are rarely re-opened. Over 66% are re-opened once and 95% fewer than five times. 7. Files re-opens are temporally related. Over 60% of re-opens occur within a minute of the first. 8. A small fraction of clients account for a large fraction of file activity. Fewer than 1% of clients account for 50% of file requests. 9. Files are infrequently shared by more than one client. Over 76% of files are never opened by more than one client. 10. File sharing is rarely concurrent and sharing is usually read-only. Only 5% of files opened by multiple clients are concurrent and 90% of sharing is read-only. 11. Most file types do not have a common access pattern.

Table 1: Summary of observations. A summary of important file system trace observations from our trace analysis. Note that we define clients to be unique IP addresses, as described in Section 4.1.

- 4) An analysis of file type and user session patterns in network file systems.
- 5) A discussion of the significant design implications derived from our study.

2 Background

In this section, we discuss previous studies, summarized in Table 2, and outline factors we believe motivate the need for new file system analysis. In addition, we provide a brief background of the CIFS protocol.

2.1 Past Studies

Early file system studies, such as those of the BSD [21], VAX/VMS [22], and Sprite [3] file systems, revealed a number of important observations and trends that guided file system design for over two decades. In particular, they observed a significant tendency towards large, sequential read access, limited read-write access, bursty I/O patterns, and very short file lifetimes. Other studies uncovered additional information such as file size distributions [18, 26] and workload self-similarity [10]. A more recent study of the Windows NT file system [29] supported a number of the past observations and trends. In 2000, Roselli, *et al.* compared four file system workloads [24]; they noted that block lifetimes had increased since past studies and explored the effect on caching strategies. The most recent study, in 2001, analyzed NFS traffic to network file servers [5], identifying a number of peculiarities with NFS tracing and arguing that pathnames can aid file system layout.

In addition to trace-based studies, which analyze file system workloads, several snapshot-based studies have analyzed file system contents [2, 4, 7, 8]. These studies showed how file attributes, such as size and type, are

distributed throughout the name-space, and also how file system contents change over time.

2.2 The Need for a New Study

Although there have been a number of previous file system trace studies, several factors that indicate that a new study may aid ongoing network file system design.

Time since last study. There have been significant changes in computing power, network bandwidth, and network file system usage since the last major study in 2001 [5]. A new study will help understand how these changes impact network file system workloads.

Few network file system studies. Only a few studies have explored network file system workloads [3, 5, 22], despite their differences from local file systems. Local file systems workloads include the access patterns of many system files, which are generally read-only and sequential, and are focused on the client's point of view. While such studies are useful for understanding client workload, it is critical for network file systems to focus on the workload seen at the server, which often excludes system files or accesses that hit the client cache.

No CIFS protocol studies. The only major network file system study in the past decade analyzed NFSv2 and v3 workloads [5]. Though NFS is common on UNIX systems, most Windows systems use CIFS. Given the widespread Windows client population and differences between CIFS and NFS (*e. g.*, CIFS is a stateful protocol, in contrast to NFSv2 and v3), analysis beyond NFS can add more insight into network file system workloads.

Limited file system workloads. University [3, 5, 10, 21, 24] and personal computer [2, 4, 32] workloads have been the focus of a number of past studies. While useful, these workloads may differ from the workloads of network file systems deployed in other environments.

Study	Date of Traces	FS/Protocol	Network FS	Trace Approach	Workload
Ousterhout, <i>et al.</i> [21]	1985	BSD		Dynamic	Engineering
Ramakrishnan, <i>et al.</i> [22]	1988-89	VAX/VMS	x	Dynamic	Engineering, HPC, Corporate
Baker, <i>et al.</i> [3]	1991	Sprite	x	Dynamic	Engineering
Gribble, <i>et al.</i> [10]	1991-97	Sprite, NFS, VxFS	x	Both	Engineering, Backup
Douceur and Bolosky [4]	1998	FAT, FAT32, NTFS		Snapshots	Engineering
Vogels [29]	1998	FAT, NTFS		Both	Engineering, HPC
Zhou and Smith [32]	1999	VFAT		Dynamic	PC
Roselli <i>et al.</i> [24]	1997-00	VxFS, NTFS		Dynamic	Engineering, Server
Ellard, <i>et al.</i> [5]	2001	NFS	x	Dynamic	Engineering, Email
Agrawal, <i>et al.</i> [2]	2000-2004	FAT, FAT32, NTFS		Snapshots	Engineering
Leung, <i>et al.</i>	2007	CIFS	x	Dynamic	Corporate, Engineering

Table 2: Summary of major file system studies over the past two decades. For each study, we show the date of trace data, the file system or protocol studied, whether it involved network file systems, the trace methodology, and the workloads studied. Dynamic trace studies involve traces of live requests. Snapshot trace studies involve snapshots of file system contents.

2.3 The CIFS Protocol

The CIFS protocol, which is based on the Server Message Block (SMB) protocol that defines most of the file transfer operations used by CIFS, differs in a number of respects from oft-studied NFSv2 and v3. Most importantly, CIFS is stateful: CIFS user and file operations act on stateful session IDs and file handles, making analysis of access patterns simpler and more accurate than in NFSv2 and v3, which require heuristics to infer the start and end of an access [5]. Although CIFS may differ from other protocols, we believe our observations are *not* tied exclusively to CIFS because access patterns, file sharing, and other workload characteristics observed by the file server are influenced by the file system users, their applications, and the behavior of the file system client, none of which are closely tied to the transfer protocol.

3 Tracing Methodology

We collected CIFS network traces from two large-scale, enterprise-class file servers deployed in the NetApp corporate headquarters. One is a mid-range file server with 3 TB of total storage, with almost 3 TB used, deployed in the corporate data center that hosts data used by over 1000 marketing, sales, and finance employees. The other is a high-end file server with over 28 TB of total storage, with 19 TB used, deployed in the engineering data center. It is used by over 500 engineering employees. Throughout the rest of this paper, we refer to these workloads as *corporate* and *engineering*, respectively.

All NetApp storage servers support multiple protocols including CIFS, NFS, iSCSI, and Fibre Channel. We traced *only* CIFS on each file server. For the corporate server, CIFS was the primary protocol used, while the engineering server saw a mix of CIFS and NFS protocols. These servers were accessed by desktops and laptops running primarily Windows for the corporate environment and a mix of Windows and Linux for the engi-

neering environment. A small number of clients also ran Mac OS X and FreeBSD. Both servers could be accessed through a Gigabit Ethernet LAN, a wireless LAN, or via a remote VPN.

Our traces were collected from both the corporate and engineering file servers between August 10th and December 14th, 2007. For each server, we mirrored a port on the network switch to which it was connected and attached it to a Linux workstation running `tcpdump` [28]. Since CIFS often utilizes NetBIOS [1], the workstation recorded all file server traffic on the NetBIOS name, datagram, and session service ports as well as traffic on the CIFS port. The trace data was periodically copied to a separate file server. Approximately 750 GB and 1.5 TB of uncompressed `tcpdump` traces were collected from the corporate and engineering servers, respectively. All traces were post-processed with `tshark 0.99.6` [30], a network packet analyzer. All filenames, usernames, and IP addresses were anonymized.

Our analysis of CIFS presented us with a number of challenges. CIFS is a stream-based protocol, so CIFS headers do not always align with TCP packet boundaries. Instead, CIFS relies on NetBIOS to define the length of the CIFS command and data. This became a problem during peak traffic periods when `tcpdump` dropped a few packets, occasionally causing a NetBIOS session header to be lost. Without the session header, `tshark` was unable to locate the beginning of the next CIFS packet within the TCP stream, though it was able to recover when it found a new session header aligned with the start of a TCP packet. To recover CIFS requests that fell within this region, we wrote a program to parse the `tcpdump` data and extract complete CIFS packets based on a signature of the CIFS header while ignoring any NetBIOS session information.

Another issue we encountered was the inability to correlate CIFS sessions to usernames. CIFS is a session-based protocol in which a user begins a session by connecting to the file server via an authenticated login pro-

cess. However, authentication in our environment almost always uses Kerberos [20]. Thus, regardless of the actual user, user authentication credentials are cryptographically changed with each login. As a result, we were unable to match a particular user across multiple sessions. Instead we relied on the client's IP address to correlate users to sessions. While less accurate, it provides a reasonable estimate of users since most users access the servers via the LAN with the same IP address.

4 Trace Analysis

This section presents the results of our analysis of our corporate and engineering CIFS workloads. We first describe the terminology used throughout our analysis. Our analysis begins with a comparison of our workloads and then a comparison to past studies. We then analyze workload activity, with a focus on I/O and sharing distributions. Finally, we examine properties of file type and user session access patterns. We italicize our key observations following the section in which they are discussed.

4.1 Terminology

Our study relies on several frequently used terms to describe our observations. Thus, we begin by defining the following terms:

I/O A single CIFS read or write command.

Sequential I/O An I/O that immediately follows the previous I/O to a file within an open/close pair (i.e., its offset equals the sum of the previous I/O's offset and length). The first I/O to an open file is always considered sequential.

Random I/O An I/O that is not sequential.

Sequential Run A series of sequential I/Os. An opened file may have multiple sequential runs.

Sequentiality Metric The fraction of bytes transferred sequentially. This metric was derived from a similar metric described by Ellard, *et al.* [5].

Open Request An open request for a file that has at least one subsequent I/O and for which a close request was observed. Some CIFS metadata operations cause files to be opened without ever being read or written. These open requests are artifacts of the CIFS client implementation, rather than the workload, and are thus excluded.

Client A unique IP address. Since Kerberos authentication prevents us from correlating usernames to users, we instead rely on IP address to identify unique clients.

	Corporate	Engineering
Clients	5261	2654
Days	65	97
Data read (GB)	364.3	723.4
Data written (GB)	177.7	364.4
R:W I/O ratio	3.2	2.3
R:W byte ratio	2.1	2.0
Total operations	228 million	352 million
Operation name	%	%
Session create	0.4	0.3
Open	12.0	11.9
Close	4.6	5.8
Read	16.2	15.1
Write	5.1	6.5
Flush	0.1	0.04
Lock	1.2	0.6
Delete	0.03	0.006
File stat	36.7	42.5
Set attribute	1.8	1.2
Directory read	10.3	11.8
Rename	0.04	0.02
Pipe transactions	1.4	0.2

Table 3: Summary of trace statistics. File system operations broken down by workload. All operations map to a single CIFS command except for file stat (composed of *query_path.info* and *query_file.info*) and directory read (composed of *find_first2* and *find_next2*). Pipe transactions map to remote IPC operations.

4.2 Workload Comparison

Table 3 shows a summary comparison of overall characteristics for both corporate and engineering workloads. For each workload we provide some general statistics along with the frequency of each CIFS request. Table 3 shows that engineering has a greater number of requests, due to a longer tracing period, though, interestingly, both workloads have similar request percentages. For both, about 21% of requests are file I/O and about 50% are metadata operations. There are also a number of CIFS-specific requests. Our I/O percentages differ from NFS workloads, in which 30–80% of all operations were I/O [5, 27]. This difference can likely attributed to both differences in workload and protocol.

Total data transferred in the two traces combined was just over 1.6 TB of data, which is less than 10% of the file servers' active storage of over 22 TB of data. Since the data transfer summaries in Table 3 include files that were transferred multiple times, our observations show that somewhat more than 90% of the active storage on the file servers was untouched over the three month trace period.

Read/write byte ratios have decreased significantly compared to past studies [3, 5, 24]. We found only a 2:1 ratio, in contrast to past studies that found ratios of 4:1 or higher, indicating workloads are becoming less read-centric. We believe that a key reason for the decrease in the read-write ratio is that client caches absorb a significant percentage of read requests. It is also interesting

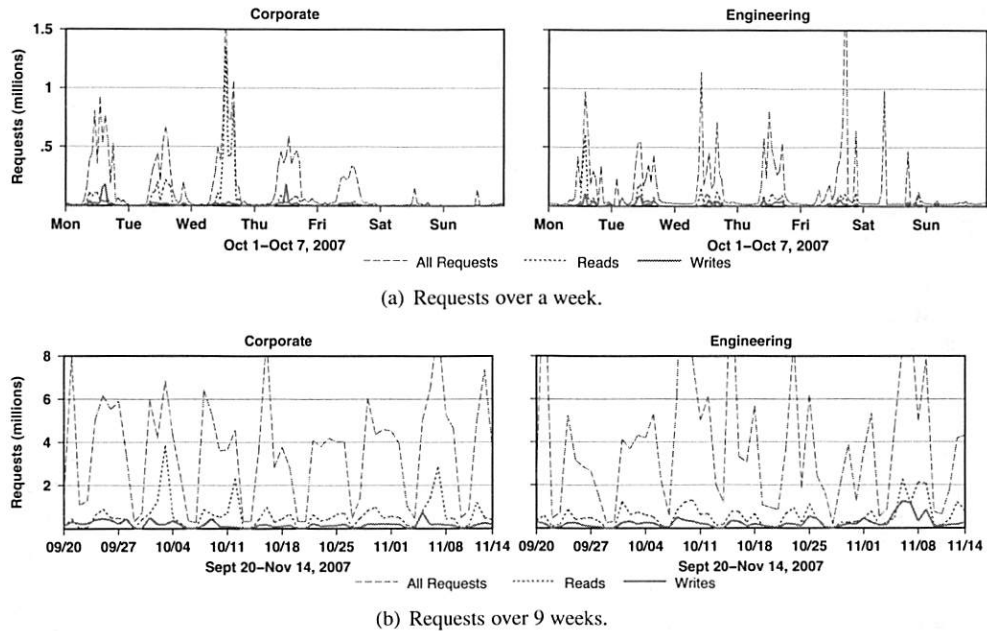


Figure 1: Request distribution over time. The frequency of all requests, read requests, and write requests are plotted over time. Figure 1(a) shows how the request distribution changes for a single week in October 2007. Here request totals are grouped in one hour intervals. The peak one hour request total for corporate is 1.7 million and 2.1 million for engineering. Figure 1(b) shows the request distribution for a nine week period between September and November 2007. Here request totals are grouped into one day intervals. The peak one day intervals are 9.4 million for corporate and 19.1 million for engineering.

that the corporate and engineering request mix are similar, perhaps because of similar work being performed on the respective clients (*e. g.*, Windows office workloads) or because client caching and I/O scheduling obfuscate the application and end-user behavior. **Observation 1** Both of our workloads are more write-heavy than workloads studied previously.

Figures 1(a) and 1(b) show the distribution of total CIFS requests and I/O requests for each workload over a one week period and a nine week period, respectively. Figure 1(a) groups request counts into hourly intervals and Figure 1(b) uses daily intervals. Figure 1(a) shows, unsurprisingly, that both workloads have strongly diurnal cycles and that there are very evident peak and idle periods throughout a day. The cyclic idle periods show there is opportunity for background processes, such as log-cleaning and disk scrubbing to run without interfering with user requests.

Interestingly, there is a significant amount of variance between individual days in the number and ratio of both requests and I/O. In days where the number of total requests are increased, the number of read and write requests are not necessarily increased. This is also the case between weeks in Figure 1(b). The variation between total requests and I/O requests implies any that single day or week is likely an insufficient profile of the overall workload, so it is probably inaccurate to extrapolate trace observations from short time periods to longer pe-

riods, as was also noted in past studies [5, 10, 29]. It is interesting to note that the overall request mix presented in Table 3 is different from the mix present in any single day or week, suggesting that the overall request mix might be different if a different time period were traced and is influenced more by workload than by behavior of the file system client.

4.3 Comparison to Past Studies

In this subsection, we compare our CIFS network file system workloads with those of past studies, including those in NFS [5], Sprite [3], VxFS [24] and Windows NT [29] studies. In particular, we analyze how file access patterns and file lifetimes have changed. For comparison purposes, we use tables and figures consistent with those of past studies.

4.3.1 File Access Patterns

Table 4 provides a summary comparison of file access patterns, showing access patterns in terms of both I/O requests and bytes transferred. Access patterns are categorized by whether a file was accessed read-only, write-only, or read-write. Sequential access is divided into two categories, *entire* accesses, which transfer the entire file, and *partial* accesses, which do not.

File System Type	Network							Local		
Workload	Corporate		Engineering		CAMPUS	EECS	Sprite	Ins	Res	NT
Access Pattern	I/Os	Bytes	I/Os	Bytes	Bytes	Bytes	Bytes	Bytes	Bytes	Bytes
Read-Only (% total)	39.0	52.1	50.6	55.3	53.1	16.6	83.5	98.7	91.0	59.0
Entire file sequential	13.5	10.5	35.2	27.4	47.7	53.9	72.5	86.3	53.0	68.0
Partial sequential	58.4	69.2	45.0	55.0	29.3	36.8	25.4	5.9	23.2	20.0
Random	28.1	20.3	19.8	17.6	23.0	9.3	2.1	7.8	23.8	12.0
Write-Only (% total)	15.1	25.2	17.3	23.6	43.8	82.3	15.4	1.1	2.9	26.0
Entire file sequential	21.2	36.2	15.6	35.2	37.2	19.6	67.0	84.7	81.0	78.0
Partial sequential	57.6	55.1	63.4	61.0	52.3	76.2	28.9	9.3	16.5	7.0
Random	21.2	8.7	21.0	3.8	10.5	4.1	4.0	6.0	2.5	15.0
Read-Write (% total)	45.9	22.7	32.1	21.1	3.1	1.1	1.1	0.2	6.1	15.0
Entire file sequential	7.4	0.1	0.4	0.1	1.4	4.4	0.1	0.1	0.0	22.0
Partial sequential	48.1	78.3	27.5	50.0	0.9	1.8	0.0	0.2	0.3	3.0
Random	44.5	21.6	72.1	49.9	97.8	93.9	99.9	99.6	99.7	74.0

Table 4: Comparison of file access patterns. File access patterns for our corporate and engineering workloads are compared with those of previous studies. CAMPUS and EECS [5] are university NFS mail server and home directory workloads, respectively. Sprite [3], Ins and Res [24] are university computer lab workloads. NT [29] is a combination of development and scientific workloads.

Table 4 shows a remarkable increase in the percentage of read-write I/O and bytes transferred. Most previous studies observed less than 7% of total bytes transferred to files accessed read-write. However, we find that both our corporate and engineering workloads have over 20% of bytes transferred in read-write accesses. Furthermore, 45.9% of all corporate I/Os and 32.1% of all engineering I/Os are in read-write accesses. This shows a diversion from the read-only oriented access patterns of past workloads. When looking closer at read-write access patterns we find that sequentiality has also changed; 78.3% and 50.0% of bytes are transferred sequentially as compared to roughly 1% of bytes in past studies. However, read-write patterns are still very random relative to read-only and write-only patterns. These changes may suggest that network file systems store a higher fraction of mutable data, such as actively changing documents, which make use of the centralized and shared environment and a smaller fraction of system files, which tend to have more sequential read accesses. These changes may also suggest that the sequential read-oriented patterns for which some file systems are designed [16] are less prevalent in network file systems, and write-optimized file systems [11, 25] may be better suited. **Observation 2** *Read-write access patterns are much more frequent compared to past studies.*

4.3.2 Sequentiality Analysis

We next compared the sequential access patterns found in our workloads with past studies. A sequential run is defined as a series of sequential I/Os to a file. Figure 2(a) shows the distribution of sequential run lengths. We see that sequential runs are short for both workloads, with almost all runs shorter than 100 KB, consistent with

past studies. This suggests that file systems should continue to optimize for short sequential common-case accesses. However, Figure 2(b), which shows the distribution of bytes transferred during sequential runs, has a very different implication, indicating that many bytes are transferred in long sequential runs: between 50–80% of bytes are transferred in runs of 10 MB or less. In addition, the distribution of sequential runs for the engineering workload is long-tailed, with 8% of bytes transferred in runs longer than 400 MB. Interestingly, read-write sequential runs exhibit very different characteristics from read-only and write-only runs: most read-write bytes are transferred in much smaller runs. This implies that the interactive nature of read-write accesses is less prone to very large transfers, which tend to be mostly read-only or write-only. Overall, we found that most bytes are transferred in much larger runs—up to 1000 times longer—when compared to those observed in past studies, though most runs are short. Our results suggest file systems must continue to optimize for small sequential access, though they must be prepared to handle a small number of very large sequential accesses. This also correlates with the heavy-tailed distributed of file sizes, which we discuss later; for every large sequential run there must be at least one large file. **Observation 3** *Bytes are transferred in much longer sequential runs than in previous studies.*

We now examine the relationship between request sizes and sequentiality. In Figure 3(a) and Figure 3(b) we plot the number of bytes transferred from a file against the sequentiality of the transfer. This is measured using the sequentiality metric: the fraction of bytes transferred sequentially, with values closer to one meaning higher sequentiality. Figures 3(a) and 3(b) show this information in a heat map in which darker regions indicate a higher fraction of transfers with that sequentiality met-

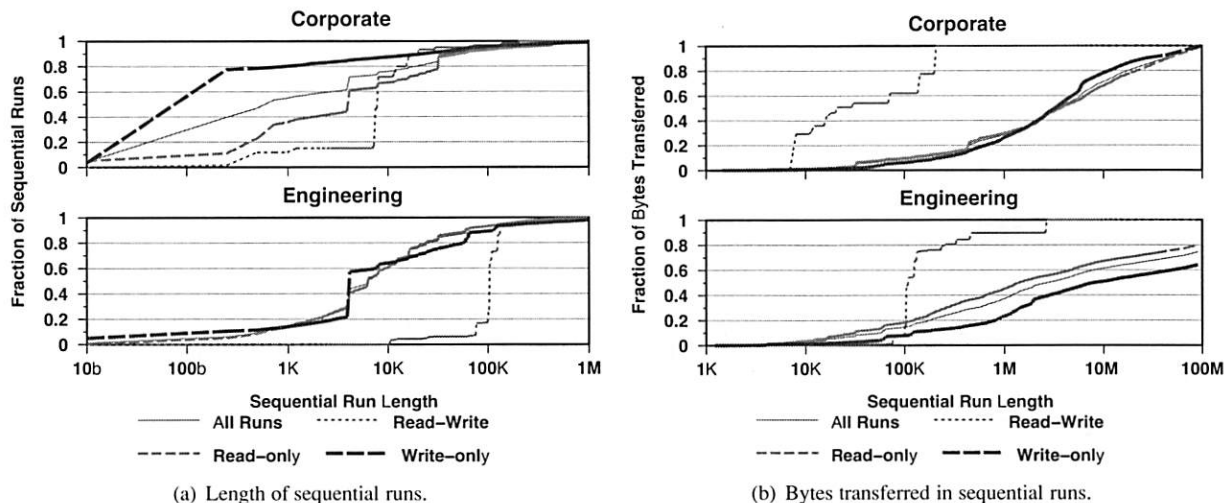


Figure 2: Sequential run properties. Sequential access patterns are analyzed for various sequential run lengths. Figure 2(a) shows the length of sequential runs, while Figure 2(b) shows how many bytes are transferred in sequential runs.

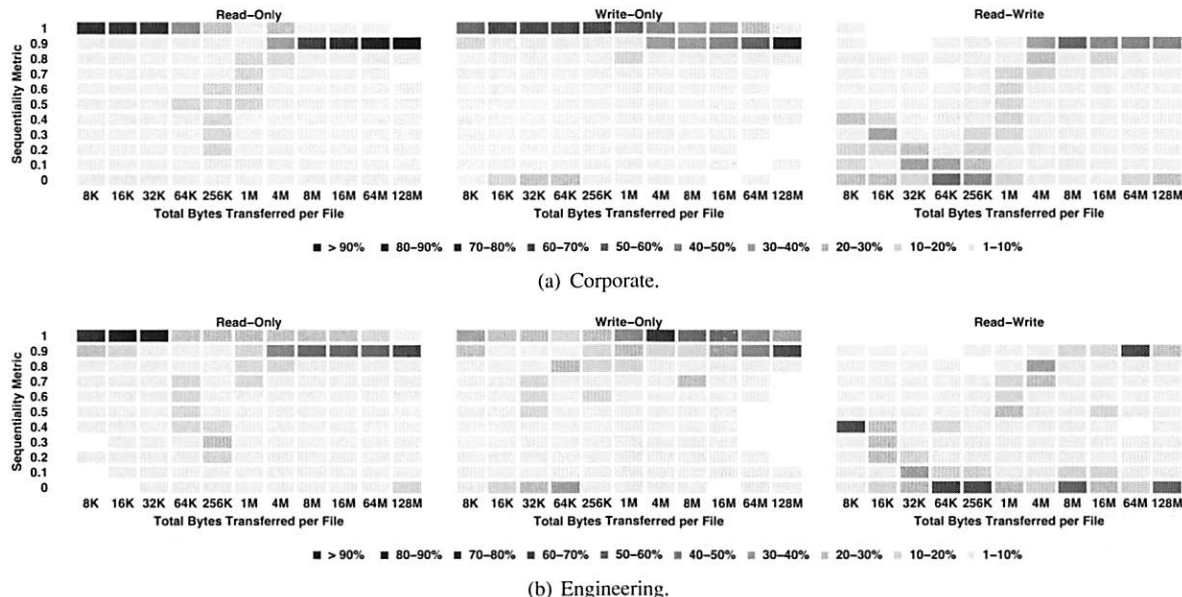
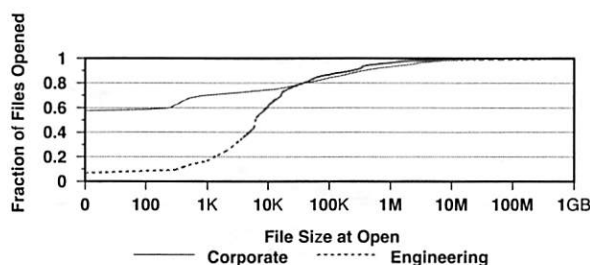


Figure 3: Sequentiality of data transfer. The frequency of sequentiality metrics is plotted against different data transfer sizes. Darker regions indicate a higher fraction of total transfers. Lighter regions indicate a lower fraction. Transfer types are broken into read-only, write-only, and read-write transfers. Sequentiality metrics are grouped by tenths for clarity.

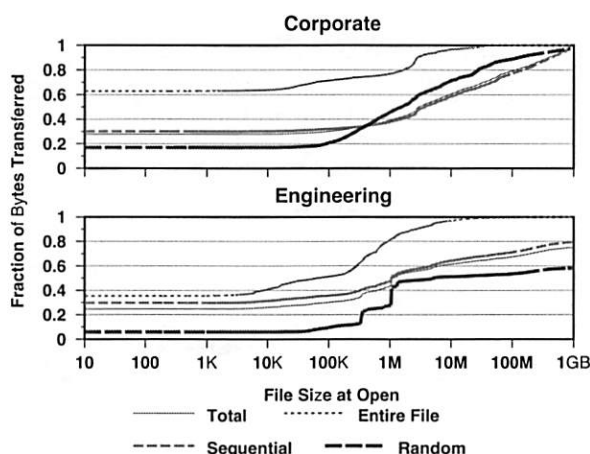
ric and lighter regions indicate a lower fraction. Each region within the heat map represents a 10% range of the sequentiality metric. We see from Figures 3(a) and 3(b) that small transfers and large transfers are more sequential for read-only and write-only access, which is the case for both workloads. However, medium-sized transfers, between 64 KB and 4 MB, are more random. For large and small transfers, file systems may be able to anticipate high sequentiality for read-only and write-only access. Read-write accesses, on the other hand, are much more random for most transfer sizes. Even very large read-

write transfers are not always very sequential, which follows from our previous observations in Figure 2(b), suggesting that file systems may have difficulty anticipating the sequentiality of read-write accesses.

Next, we analyze the relationship between file size and access pattern by examining the size of files at open time to determine the most frequently opened file sizes and the file sizes from which most bytes are transferred. It should be noted that since we only look at opened files, it is possible that this does not correlate to the file size distribution across the file system. Our results are shown



(a) Open requests by file size.



(b) Bytes transferred by file size.

Figure 4: File size access patterns. The distribution of open requests and bytes transferred are analyzed according to file size at open. Figure 4(a) shows the size of files most frequently opened. Figure 4(b) shows the size of files from which most bytes are transferred.

in Figures 4(a) and 4(b). In Figure 4(a) we see that 57.5% of opens in the corporate workload are to newly-created files or truncated files with zero size. However, this is not the case in the engineering workload, where only 6.3% of opens are to zero-size files. Interestingly, both workloads find that most opened files are small; 75% of opened files are smaller than 20 KB. However, Figure 4(a) shows that most bytes are transferred from much larger files. In both workloads, we see that only about 60% of bytes are transferred from files smaller than 10 MB. The engineering distribution is also long-tailed with 12% of bytes being transferred from files larger than 5 GB. By comparison, almost all of the bytes transferred in previous studies came from files smaller than 10 MB. These observations suggest that larger files play a more significant role in network file system workloads than in those previously studied. This may be due to frequent small file requests hitting the local client cache. Thus, file systems should still optimize small file layout for frequent access and large file layout for large transfers. **Observation 4 Bytes are transferred from much larger files than in previous studies.**

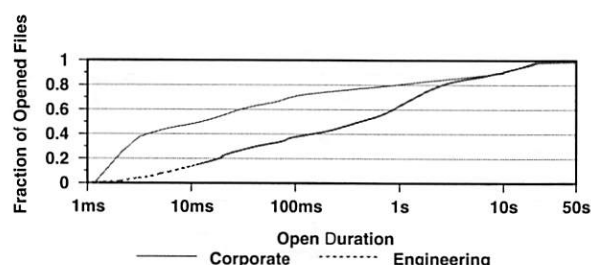


Figure 5: File open durations. The duration of file opens is analyzed. Most files are opened very briefly, although engineering files are opened slightly longer than corporate files.

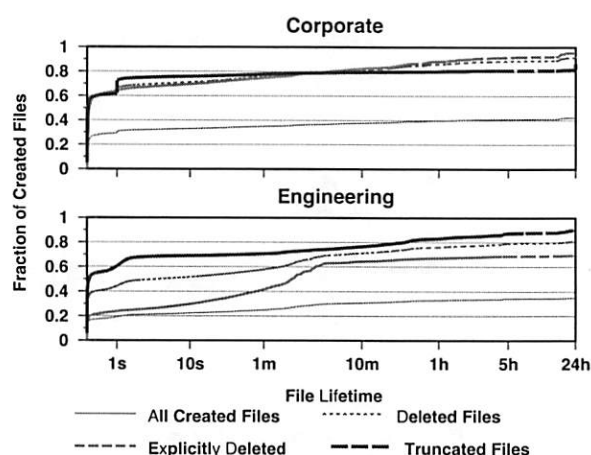


Figure 6: File lifetimes. The distributions of lifetimes for all created and deleted files are shown. Files may be deleted through explicit delete request or truncation.

Figure 5 shows the distribution of file open durations. We find that files are opened for shorter durations in the corporate workload than in the engineering workload. In the corporate workload, 71.1% of opens are shorter than 100ms, but just 37.1% are similarly short in the engineering workload. However, for both workloads most open durations are less than 10 seconds, which is similar to observations in past studies. This is also consistent with our previous observations that small files, which likely have short open durations, are most frequently accessed.

4.3.3 File Lifetime

This subsection examines how file lifetimes have changed as compared to past studies. In CIFS, files can be either deleted through an explicit delete request, which frees the entire file and its name, or through truncation, which only frees the data. Figure 6 shows the distribution of file lifetimes, broken down by deletion method. We find that most created files live longer than 24 hours, with 57.0% and 64.9% of corporate and engineering files persisting for more than a day. Both dis-

tributions are long-tailed, meaning many files live well beyond 24 hours. However, files that *are* deleted usually live less than a day: only 18.7% and 6.9% of eventually-deleted files live more than 24 hours. Nonetheless, compared to past studies in which almost all deleted files live less than a minute, deleted files in our workloads tend to live much longer. This may be due to fewer temporary files being created over the network. However, we still find some files live very short lifetimes. In each workload, 56.4% and 38.6% of deleted files are deleted within 100 ms of creation, indicating that file systems should expect fewer files to be deleted and files that live beyond than a few hundred milliseconds to have long lifetimes.

Observation 5 *Files live an order of magnitude longer than in previous studies.*

4.4 File I/O Properties

We now take a closer look at the properties of file I/O, where, as defined in Section 4.1, an I/O request is defined as any single read or write operation. We begin by looking at per-file, per-session I/O inter-arrival times, which include network round-trip latency. Intervals are categorized by the type of requests (read or write) that bracket the interval; the distribution of interval lengths is shown in Figure 7(a). We find that most inter-arrival times are between 100 μ s and 100 ms. In fact, 96.4% and 97.7% of all I/Os have arrival times longer than 100 μ s and 91.6% and 92.4% are less than 100 ms for corporate and engineering, respectively. This tight window means that file systems may be able to make informed decisions about when to prefetch or flush cache data. Interestingly, there is little distinction between read-read or read-write and write-read or write-write inter-arrival times. Also, 67.5% and 69.9% of I/O requests have an inter-arrival time of less than 3 ms, which is shorter than some measured disk response times [23]. These observations may indicate cache hits at the server or possibly asynchronous I/O. It is also interesting that both workloads have similar inter-arrival time distributions even though the hardware they are running is of different classes, a mid-range model versus a high-end model.

Next, we examine the distribution of bytes transferred by a single I/O request. As Figure 7(b) shows, most requests transfer less than 8 KB, despite a 64 KB maximum request size in CIFS. This distribution may vary between CIFS and NFS since each buffers and schedules I/O differently. The distribution in Figure 7(b) increases for only a few I/O sizes, indicating that clients generally use a few specific request sizes. This I/O size information can be combined with I/O inter-arrival times from Figure 7(a) to calculate a distribution of I/Os per-second (IOPS) that may help file systems determine how much buffer space is required to support various I/O rates.

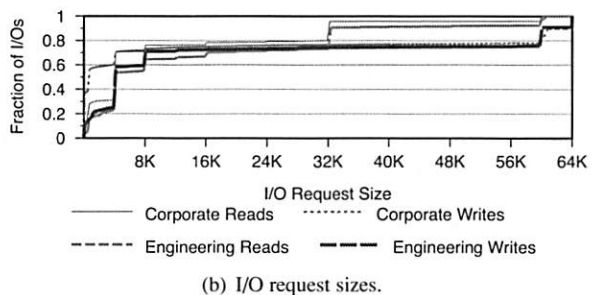
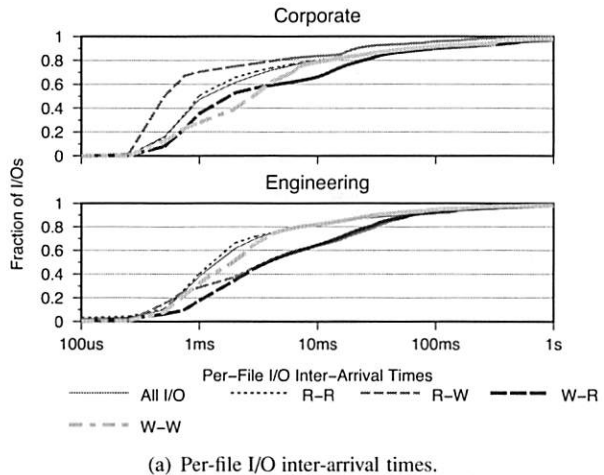


Figure 7: File I/O properties. The burstiness and size properties of I/O requests are shown. Figure 7(a) shows the I/O inter-arrival times. Figure 7(b) shows the sizes of read and write I/O.

4.5 File Re-Opens

In this subsection, we explore how frequently files are re-opened, *i. e.*, opened more than once during the trace period. Figure 8(a), shows the distribution of the number of times a file is opened. For both workloads, we find that the majority of files, 65%, are only opened once during the entire trace period. The infrequent re-access of many files suggests there are opportunities for files to be archived or moved to lower-tier storage. Further, we find that about 94% of files are accessed fewer than five times. However, both of these distributions are long-tailed—some files are opened well over 100,000 times. These frequently re-opened files account for about 7% of total opens in both workloads.

Observation 6 *Most files are not re-opened once they are closed.*

We now look at inter-arrival times between re-opens of a file. Re-open inter-arrival time is defined as the duration between the last close of a file and the time it is re-opened. A re-open is considered *concurrent* if a re-open occurs while the file is still open (*i. e.*, it has not yet been closed). The distribution of re-open inter-arrival times is shown in Figure 8(b). We see that few re-opens are concurrent, with only 4.7% of corporate re-opens and

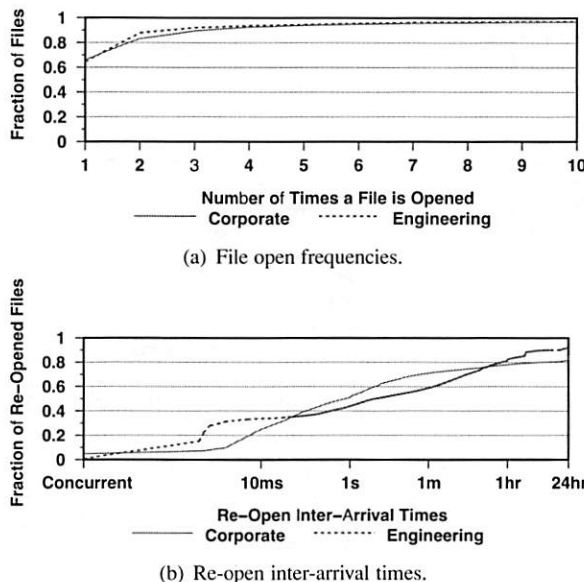


Figure 8: File open properties. The frequency of and duration between file re-opens is shown. Figure 8(a) shows how often files are opened more than once. Figure 8(b) shows the time between re-opens.

0.7% of engineering re-opens occurring on an currently-open file. However, re-opens are temporally related to the previous close; 71.1% and 58.8% of re-opens occur less than one minute after the file is closed. Using this information, file systems may be able to decide when a file should be removed from the buffer cache or when it should be scheduled for migration to another storage tier. **Observation 7** If a file is re-opened, it is temporally related to the previous close.

4.6 Client Request Distribution

We next examine the distribution of file open and data requests amongst clients; recall from Section 4.1 that “client” refers to a unique IP address rather than an individual user. We use Lorenz curves [14]—cumulative distribution functions of probability distributions—rather than random variables to show the distribution of requests across clients. Our results, shown in Figure 9, find that a tiny fraction of clients are responsible for a significant fraction of open requests and bytes transferred. In corporate and engineering, 0.02% and 0.04% of clients make 11.9% and 22.5% of open requests and account for 10.8% and 24.6% of bytes transferred, respectively. Interestingly, 0.02% of corporate clients and 0.04% of engineering clients correspond to approximately 1 client for each workload. Additionally, we find that about 35 corporate clients and 5 engineering clients account for close to 50% of the opens in each workload. This suggests that the distribution of activity is highly skewed and

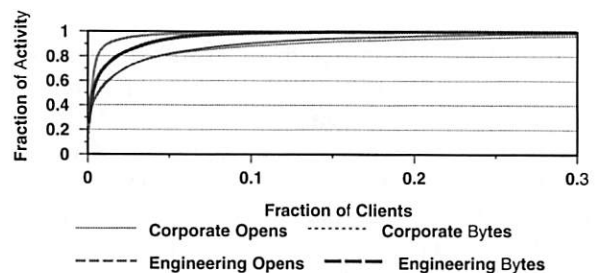


Figure 9: Client activity distribution The fraction of clients responsible for certain activities is plotted.

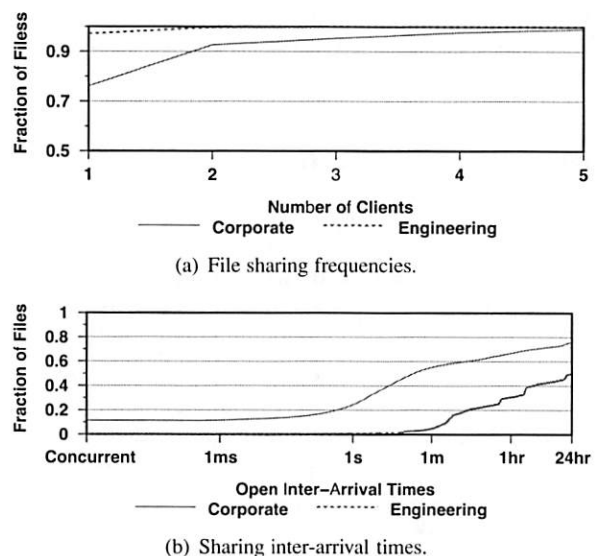


Figure 10: File sharing properties. We analyze the frequency and temporal properties of file sharing. Figure 10(a) shows the distribution of files opened by multiple clients. Figure 10(b) show the duration between shared opens.

that file systems may be able to take advantage of this information for doing informed allocation of resources or quality of service planning. **Observation 8** A small fraction of clients account for a large fraction of file activity.

4.7 File Sharing

This subsection looks at the extent of file sharing in our workloads. A file is shared when two or more clients open the same file at some time during the trace period; the sharing need not be concurrent. Since we can only distinguish IP addresses and not actual users, it is possible that two IP addresses may represent a single (human) user and vice versa. However, the drastic skew of our results indicates this likely has little impact on our observations. Also, we only consider opened files in our analysis; files which have only had their metadata accessed by multiple clients are not included in these results.

Figure 10(a) shows the distribution of the frequency with which files are opened by multiple clients. We find that most files are only opened by a single client. In fact, 76.1% and 97.1% of files are only opened by one client in corporate and engineering, respectively. Also, 92.7% and 99.7% of files are ever opened by two or fewer clients. This suggests that the shared environment offered by network file systems is not often taken advantage of. Other methods of sharing files, such as email or web and wiki pages, may reduce the need for clients to share files via the file system. However, both distributions are long-tailed, and a few files are opened by many clients. In the corporate workload, four files are opened by over 2,000 clients and in the engineering workload, one file is opened by over 1,500 clients. This shows that, while not common, sharing files through the file system can be heavily used on occasion. **Observation 9** Files are infrequently accessed by more than one client.

In Figure 10(b) we examine inter-arrival times between different clients opening a file. We find that concurrent (simultaneous) file sharing is rare. Only 11.4% and 0.2% of shared opens from different clients were concurrent in corporate and engineering, respectively. When combined with the observation that most files are only opened by a single client, this suggests that synchronization for shared file access is not often required, indicating that file systems may benefit from looser locking semantics. However, when examining the duration between shared opens we find that sharing does have a temporal relationship in the corporate workload; 55.2% of shared opens occur within one minute of each other. However, this is not true for engineering, where only 4.9% of shared opens occur within one minute.

We now look at the manner (read-only, write-only, or read-write) with which shared files are accessed. Figure 11 shows the usage patterns for files opened by multiple clients. Gaps are present where no files were opened by that number of clients. We see that shared files are accessed read-only the majority of the time. These may be instances of reference documents or web pages that are rarely re-written. The number of read-only accesses slightly decreases as more clients access a file and a read-write pattern begins to emerge. This suggests that files accessed by many clients are more mutable. These may be business documents, source code, or web pages. Since synchronization is often only required for multiple concurrent writers, these results further argue for loose file system synchronization mechanisms. **Observation 10** File sharing is rarely concurrent and mostly read-only.

Finally, we analyze which clients account for the most opens to shared files. Equality measures how open requests are distributed amongst clients sharing a file. Equal file sharing implies all sharing clients open the shared file an equal number of times. To analyze equal-

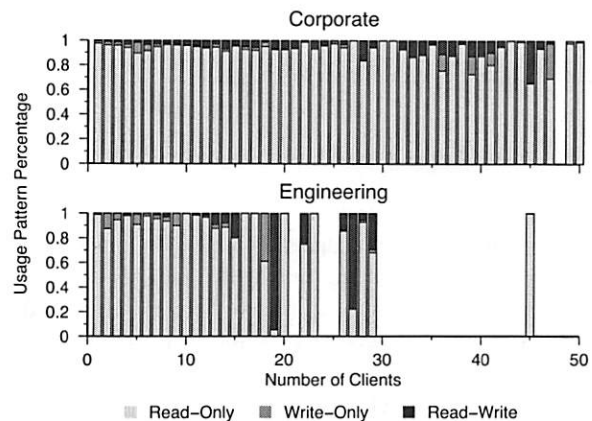


Figure 11: File sharing access patterns. The fraction of read-only, write-only, and read-write accesses are shown for differing numbers of sharing clients. Gaps are seen where no files were shared with that number of clients.

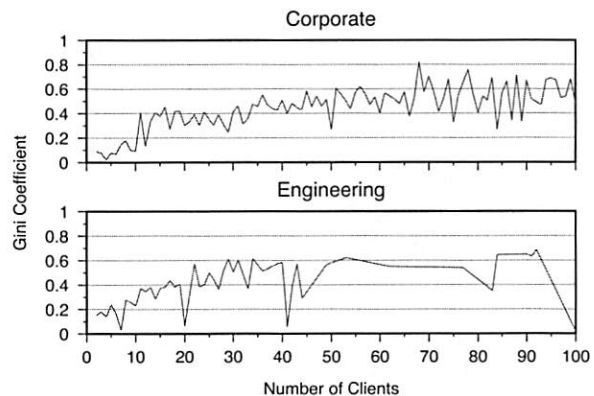


Figure 12: User file sharing equality. The equality of sharing is shown for differing numbers of sharing clients. The Gini coefficient, which measures the level of equality, is near 0 when sharing clients have about the same number of opens to a file. It is near 1 when clients unevenly share opens to a file.

ity, we use the Gini coefficient [9], which measures statistical dispersion, such as the inequality of income in economic analysis. We apply the equality concept to how frequently a shared file is opened by a client. Lower coefficients mean sharing clients open the file more equally (the same number of times), and higher coefficients mean a few clients account for the majority of opens. Figure 12 shows Gini coefficients for various numbers of shared clients. We see that, as more clients open a file, the level of equality decreases, meaning few clients begin to dominate the number of open requests. Gini coefficients are lower, less than 0.4, for files opened by fewer than 20 clients, meaning that when a few clients access a file, they each open the file an almost equal number of times. As more clients access the file, a small number of clients begin to account for most of the opens. This may indicate

that as more clients share a file, it becomes less reasonable for all sharing clients to access the file evenly, and a few dominate clients begin to emerge.

4.8 File Type and User Session Patterns

There have been a number of attempts to make layout, caching, and prefetching decisions based on how specific file types are accessed and the access patterns of certain users [6, 17]. In this subsection, we take a closer at how certain file types are accessed and the access patterns that occur between when a user begins a CIFS “user session” by logging on and when they log-off. Our emphasis is on whether file types or users have common access patterns that can be exploited by the file system. We begin our analysis by breaking down file type frequencies for both workloads. Figures 13(a) and 13(b) show the most frequently opened and most frequently read and written file types. For frequently read and written file types, we show the fraction of bytes read for that type. Files with no discernible file extension are labeled “unknown.”

We find that the corporate workload has no file type, other than unknown types, that dominates open requests. However, 37.4% of all opens in the engineering workload are for C header files. Both workloads have a single file type that consumes close to 20% of all read and write I/O. Not surprisingly, these types correspond to generally large files, *e. g.*, mdb (Microsoft Access Database) files and vmdk (VMWare Virtual Disk) files. However, we find that most file types do not consume a significantly large fraction of open or I/O requests. This shows that file systems likely can be optimized for the small subset of frequently accessed file types. Interestingly, there appears to be little correlation between how frequently a file is opened and how frequently it is read or written. Only three corporate and two engineering file types appear as both frequently opened and frequently read or written; the mdb and vmdk types only constitute 0.5% and 0.08% of opens. Also, it appears file types that are frequently read or written are mostly read.

We now analyze the hypothesis that file systems can make use of file type and user access patterns to improve layout and prefetching [5, 6, 17]. We do so examining *access signatures*, a vector containing the number of bytes read, bytes written, and sequentiality metric of a file access. We start by defining an access signature for each open/close pair for each file type above, we then apply K-means clustering [15] to the access signatures of each file type. K-means groups access signatures with similar patterns into unique clusters with varying densities. Our results are shown in Figure 14. For clarity we have categorized access signatures by the access type: read-only, write-only, or read-write. We further group signatures by their sequentiality metric ranges: 1–0.81 is consid-

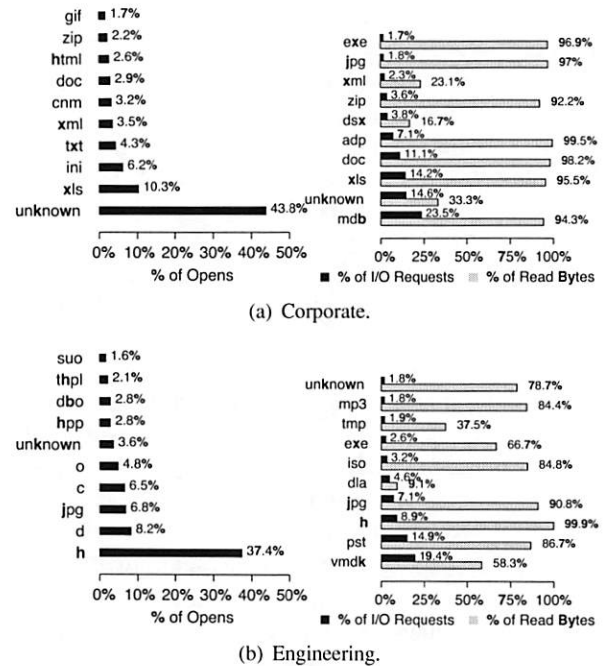


Figure 13: File type popularity. The histograms on the right show which file types are opened most frequently. Those on the left show the file types most frequently read or written and the percentage of accessed bytes read for those types.

ered highly sequential, 0.8–0.2 is considered mixed sequentiality, and 0.19–0 is considered highly random. Finally, access signatures are categorized by the number of bytes transferred; access signatures are considered small if they transfer no more than 100 KB and large otherwise. Darker regions indicate the file type has a higher fraction of access signatures with those properties shown on the *y*-axis, and lighter regions indicate fewer signatures with those characteristics.

Figure 14 shows that most file types have several distinct kinds of access patterns, rather than one as previously presumed. We also see that each type has multiple patterns that are more frequent than others, suggesting that file systems may not be able to properly predict file type access patterns using only a single pattern. Interestingly, small sequential read patterns occur frequently across most of the file types, implying that file systems should be optimized for this pattern, as is often already done. **Observation 11** *Most file types do not have a single pattern of access.*

Surprisingly, file types such as vmdk that consume a large fraction of total I/Os are frequently accessed with small sequential reads. In fact, 91% of all vmdk accesses are of this pattern, contradicting the intuition derived from Figure 13(b) that vmdk files have large accesses. However, a much smaller fraction of vmdk accesses transfer huge numbers of bytes in highly random read-write patterns. Several patterns read and write over

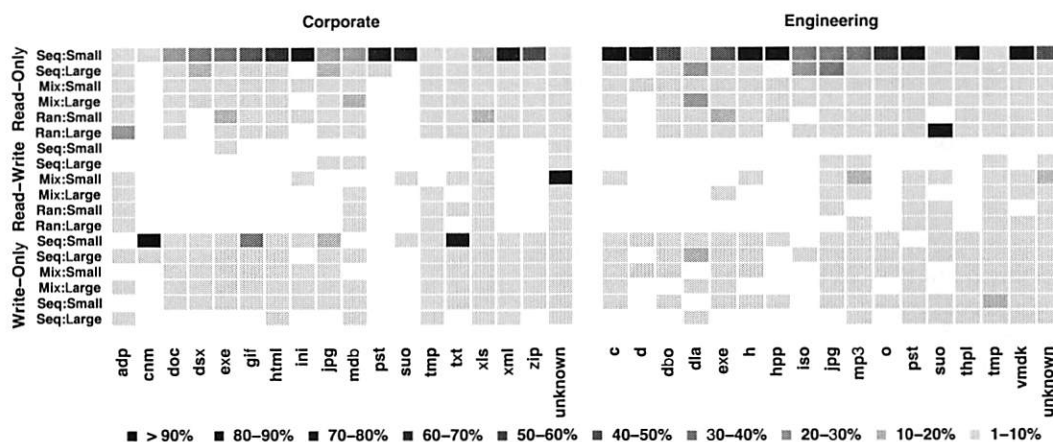


Figure 14: File type access patterns The frequency of access patterns are plotted for various file types. Access patterns are categorized into 18 groups. Increasingly dark regions indicate higher fractions of accesses with that pattern.

10 GB of data with a sequentiality metric less than 0.5, showing that frequent patterns may not be representative of the significant patterns in terms of bytes transferred or sequentiality. This argues that file systems should anticipate several patterns of access for any file type if layout or prefetching benefits are to be gained. Also, it is critical that they identify transitions between patterns. For example, a file system may, by default, prefetch data for vmdk files in small chunks: 100 KB or less. However, when over 100 KB of a vmdk file is accessed this signals the likely start of a very large transfer. In this case, the file system must properly adjust its prefetching.

Our observation that many file types exhibit several access patterns of varying frequency and significance draws an interesting comparison to the results in Table 4. Table 4 shows significant read-write I/O and byte transfer activity. However, file types in Figure 14 rarely have read-write patterns. This implies, read-write file accesses are, in general, uncommon, however when they do occur, a large number of bytes are accessed.

Next, we apply the same K-means clustering approach to access signatures of access patterns that occur within a CIFS user session. Recall that CIFS users begin a connection to the file server by creating an authenticated user session and end by eventually logging off. We define signatures for all accesses performed while the user is logged on. However, we only consider sessions in which bytes are transferred. The CIFS client opens short, temporary sessions for various auxiliary functions, which we exclude from this study as they do not represent a normal user log-in. Like file types, user sessions have several common patterns and no single pattern can summarize all of a user's accesses. The majority of user sessions have read-write patterns with less than 30 MB read and 10 MB written with a sequentiality metric close to 0.5, while a few patterns have much more significant data transfers that read and write gigabytes of data.

5 Design Implications

In this section we explore some of the possible implications of our trace analysis on network file system designs. We found that read-write access patterns have significantly increased relative to previous studies (see Section 4.3.1). Though we observed higher sequentiality in read-write patterns than past studies, they are still highly random compared to read-only or write-only access patterns (see Section 4.3.1). In contrast, a number of past studies found that most I/Os and bytes are transferred in read-only sequential access patterns [3, 21, 29], which has impacted the designs of several file systems [16, 19]. Our observed shift towards read-write access patterns suggests file systems should look towards improving random access performance, perhaps through alternative media, such as flash. In addition, we observed that the ratio of data read to data written is decreasing compared to past studies [3, 5, 24] (see Section 4.2). This decrease is likely due to increasing effectiveness of client caches and fewer read-heavy system files on network storage. When coupled with increasing read-write access patterns, write-optimized file systems, such as LFS [25] and WAFL [11], or NVRAM write caching appear to be good designs for network file systems.

We also observed that files are infrequently re-opened (see Section 4.5) and are usually accessed by only one client (see Section 4.7). This suggests that caching strategies which exploit this, such as exclusive caching [31], may have practical benefits. Also, the limited reuse of files indicates that increasing the size of server data caches may add only marginal benefit; rather, file servers may find larger metadata caches more valuable because metadata requests made up roughly 50% of all operations in both workloads, as Section 4.2 details.

Our finding that most created files are not deleted (see Section 4.3.3) and few files are accessed more than once

(see Section 4.5) suggests that many files may be good candidates for migration to lower-tier storage or archives. This is further motivated by our observation that only 1.6TB were transferred from 22TB of in use storage over three months. While access to file metadata should be fast, this indicates much file data can be compressed, de-duplicated, or placed on low power storage, improving utilization and power consumption, without significantly impacting performance. In addition, our observation that file re-accesses are temporally correlated (see Section 4.5) means there are opportunities for intelligent migration scheduling decisions.

6 Conclusions

In this paper we presented an analysis of two large-scale CIFS network file system workloads gathered from enterprise-class file servers deployed in a corporate and in an engineering environment. We compared our workloads to previous file system studies to understand how file access patterns have changed and conducted a number of other experiments. We found that read-write file access patterns and random file access are far more common than previously thought, and that most file storage remains unused, even over a three month period. Our observations on sequentiality, file lifetime, file reuse and sharing, and request type distribution also differ from those in earlier studies. Based on these observations, we made several recommendations for improving network file server design to handle the workload of modern corporate and engineering environments.

Acknowledgments

This work was supported in part by the Department of Energy under award DE-FC02-06ER25768, the NSF under award CCF-0621463, and industrial sponsors of the Storage Systems Research Center at UC Santa Cruz, including Agami Systems, Data Domain, Hewlett Packard, LSI Logic, NetApp, Seagate, and Symantec. We would also like to thank our colleagues in the SSRC and NetApp's Advanced Technology Group, our shepherd Jason Flinn, and the anonymous reviewers for their insightful feedback, which greatly improved the quality of the paper.

References

- [1] A. Aggarwal and K. Auerbach. Protocol standard for a netbios service on a tcp/udp transport. IETF Network Working Group RFC 1001, March 1987.
- [2] N. Agrawal, *et al.* A five-year study of file-system metadata. In *Proc. of FAST '07*, Feb. 2007.

- [3] M. G. Baker, *et al.* Measurements of a distributed file system. In *Proc. SOSP '91*, Oct. 1991.
- [4] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proc. of SIGMETRICS '99*, 1999.
- [5] D. Ellard, *et al.* Passive NFS tracing of email and research workloads. In *Proc. of FAST '03*, 2003.
- [6] D. Ellard, *et al.* Attribute-based prediction of file properties. Technical Report TR-14-03, Harvard, 2004.
- [7] K. Evans and G. H. Kuenning. A study of irregularities in file-size distributions. In *Proceedings of SPECTS '02*.
- [8] T. J. Gibson and E. L. Miller. Long-term file activity patterns in a UNIX workstation environment. In *Proc. of the 15th IEEE Symposium on Mass Storage Systems*, pages 355–372, Mar. 1998.
- [9] C. Gini. Measurement of inequality and incomes. *The Economic Journal*, 31:124–126, 1921.
- [10] S. Gribble, *et al.* Self-similarity in file systems: Measurement and applications. In *Proc. of SIGMETRICS '98*.
- [11] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proc. of USENIX '94*.
- [12] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM ToCS*, 10(1), 1992.
- [13] P. J. Leach and D. C. Naik. A common internet file system (cifs/1.0) protocol. IETF Network Working Group RFC Draft, March 1997.
- [14] M. O. Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association*, 9:209–219, 1905.
- [15] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the 5th Berkeley Symp. on Mathematical Statistics and Probability*, pages 281–297, 1967. University of California Press.
- [16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM ToCS*, 2(3), Aug. 1984.
- [17] M. Mesnier, *et al.* File classification in self-* storage systems. In *Proc. of ICAC '04*.
- [18] S. J. Mullender and A. S. Tanenbaum. Immediate files. *Software-Practice and Experience*, 14(4), April 1984.
- [19] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM ToCS*, 6(1), 1988.
- [20] B. C. Neumann, *et al.* Kerberos: An authentication service for open network systems. In *Proc. of USENIX '88*.
- [21] J. K. Ousterhout, *et al.* A trace-driven analysis of the Unix 4.2 BSD file system. In *Proc. of SOSP '85*, 1985.
- [22] K. K. Ramakrishnan, P. Biswas, and R. Karedla. Analysis of file i/o traces in commercial computing environments. In *Proc. of SIGMETRICS '92*, 1992.
- [23] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proc. of USENIX '06*, May 2006.
- [24] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proc. of USENIX '00*.
- [25] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM ToCS*, 10(1):26–52, Feb. 1992.
- [26] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proc. of SOSP '81*, Dec. 1981.
- [27] Spec benchmarks. <http://www.spec.org/benchmarks.html>.
- [28] Tcpdump/libpcap. <http://www.tcpdump.org/>.
- [29] W. Vogels. File system usage in Windows NT 4.0. In *Proc. of SOSP '99*, Dec. 1999.
- [30] Wireshark: Go deep. <http://www.wireshark.org/>.
- [31] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc. of USENIX '02*.
- [32] M. Zhou and A. J. Smith. Analysis of personal computer workloads. In *Proc. of MASCOTS '99*, 1999.

Evaluating Distributed Systems: Does Background Traffic Matter?

Kashi Venkatesh Vishwanath and Amin Vahdat

University of California, San Diego

{kvishwanath,vahdat}@cs.ucsd.edu

Abstract

Evaluating novel networked protocols and services requires subjecting the target system to realistic Internet conditions. However, there is no common understanding of what is required to capture such realism. Conventional wisdom suggests that competing background traffic will influence service and protocol behavior. Once again however, there is no understanding of what aspects of background traffic are important and the extent to which services are sensitive to these characteristics.

Earlier work shows that Internet traffic demonstrates significant burstiness at a range of time scales. Unfortunately, existing systems evaluations either do not consider background traffic or employ simple synthetic models, e.g., based on Poisson arrivals, that do not capture these burstiness properties. In this paper, we show that realistic background traffic, has qualitatively different impact on application and protocol behavior than simple traffic models. One conclusion from our work is that applications should be evaluated under a range of background traffic characteristics to determine the relative merits of applications and to understand behavior in corner cases of live deployment.

1 Introduction

There has been significant interest in understanding the characteristics of network services and protocols under realistic deployment conditions [2, 3, 4, 13]. Application developers typically have two high-level options when evaluating their prototype: live deployment or emulation/simulation. Live deployment on a testbed such as PlanetLab allows developers to subject their system to realistic network conditions and failures. However, experiment management/control and reproducibility become more difficult. Further, while the deployment environment is realistic, there is no guarantee that it is representative or that any series of experiments will experience a range of desired potential network conditions.

Emulation and simulation on the other hand simplify experiment management and make it relatively easy to obtain reproducible results. However, while recent emulation environments [15, 20] allow unmodified applications, they must still simulate some target network conditions, including topology, failure characteristics, routing, and background traffic. Practitioners are left with the un-

enviable task of developing appropriate models for each of these important network characteristics. Thus, while emulation offers the promise of evaluating applications under a range of conditions, the huge space of potential conditions makes it difficult, if not impossible, for most developers to take advantage of this flexibility.

One long term goal of our work is to enable developers to use emulation to evaluate their applications under a range of scenarios with realistic models of traffic, topology, routing, and host characteristics. We leave a study of the relative sensitivity of applications to different aspects of the network, e.g., routing protocols versus traffic characteristics [21, 19] versus topology [7, 11], to future work. Our goal in this paper is to understand application sensitivity to background traffic. It is clear that applications behave differently when competing with other traffic. Thus, it is not surprising that researchers have begun to include background traffic models in their evaluations (see Section 2 for a summary). However, it is also well-known that Internet traffic has rich and complex properties not captured by models for background traffic, e.g., self-similarity and burstiness at a range of timescales [8, 22]) not captured by the simple models employed by practitioners. A natural question to answer then, is whether these complex but realistic models, warrant attention as candidates for background traffic.

In this paper, we quantify application sensitivity to a range of background traffic characteristics. That is, are simple models of background traffic, such as constant bit rate, Poisson arrivals, or deterministic link loss rates, sufficient to capture the effects of background traffic? Or do we require more complex background traffic models that capture the burstiness on a particular network link? We begin with a literature survey to understand the common techniques for modeling background traffic. We also leverage recent work [19, 21] on modeling and recreating background traffic characteristics for existing Internet links. Using accurate, real-time network emulation, we subject a number of applications to a spectrum of background traffic models and report variations in end-to-end application behavior.

We find qualitative differences in application behavior when employing simple synthetic models of background traffic rather than realistic traffic models. We investigate the cause of this difference and present our

initial findings. *Specifically, we find that differences in burstiness between two background traffic models at a range of timescales significantly impacts overall application behavior, even when network links are not congested.* Existing synthetic models for background traffic do not demonstrate the rich variances in the packet arrival process for competing traffic present in live network traffic. Thus, studies employing these synthetic models may mischaracterize the impact of background traffic. *Further, we find that even seemingly small differences in the burstiness of background traffic for realistic traffic models can lead to important differences in application behavior.* Hence, one conclusion of this work is that studies of network application behavior should include experiments with a range of realistic background traffic models. Ideally, the research community would evolve a suite of background traffic models representative of a range of network conditions and locations. We hope that our findings in the paper will serve as a means to spur sufficient interest in the community to collectively develop such an appropriate benchmark suite in the future.

In summary, this paper makes the following contributions. This work is the first to quantify the impact of a range of background traffic characteristics on a number of applications. Prior to this work, it was not possible to deterministically subject application traffic to a range of realistic network conditions while accounting for the complexity of real network traffic, e.g., as determined by TCP. We present a methodology for doing so and use this methodology to carry out a systematic sensitivity study of applications to a range of network characteristics. We show that techniques such as replaying a pre-existing trace packet-by-packet do not exhibit the responsiveness of real Internet traffic. Similarly, we show that common models for generating background traffic, such as transmitting traffic at a constant bit rate, traffic with a Poisson arrival process, or deterministically setting loss rates to network links has significantly less impact on application traffic than realistic Internet traffic.

Investigating the cause of these observations, our detailed performance evaluation shows that the properties of Internet traffic, in particular its burstiness across a range of time scales, can have unpredictable impact on a range of applications relative to simpler traffic models. We establish that it is not enough to simply use “some” bursty source as a background traffic model. As another example, we reproduce the results of an earlier study that employed synthetic traffic models to compare bandwidth estimation tools. After validating the original results, we found that some of the conclusions of the earlier study may have been reversed when employing realistic traffic models.

2 Motivation and Related Work

Consider the problem of determining the sensitivity of a given application to a range of background traffic characteristics. One approach would simply be to run the application across the Internet on a testbed such as PlanetLab. Unfortunately, the difficulty to measure the characteristics of background traffic at any point in time is compounded by the fact that one cannot guarantee reproducing a particular set of background traffic characteristics. Finally, experiments would be restricted to the type of background traffic experienced in a particular deployment scenario, making it difficult to extrapolate to sensitivity in other settings.

Hence, a careful study of application sensitivity to background traffic must run in an emulation or simulation environment prior to live deployment. This begs the question as to what kind of background traffic to employ. One approach is to take a trace of background traffic at a particular point in the Internet and to replay that traffic packet-by-packet in an emulation environment. As we quantify later, such background traffic will not be *responsive* to the application traffic. It will blindly transmit data in a preconfigured sequence; real Internet traffic responds and adapts to prevailing traffic conditions as a result of end-to-end congestion control. Other simple approaches involve playing back traffic at a constant bit rate, according to a Poisson arrival process, or using deterministic loss rates. Unfortunately, these simple techniques are known not to reproduce the characteristics of Internet traffic and, as we quantify later, will result in incorrectly estimating the impact of background traffic.

In this paper, we present a methodology for quantifying the impact of realistic Internet traffic on a range of applications. We build on our earlier work [21] that shows how to create traffic that is both *realistic* and *responsive*. By realistic, we mean that the traffic matches the complex characteristics of traffic across some original link, including traffic burstiness at a range of timescales. By responsive, we mean that the background traffic adapts to application traffic in the same manner that they would in the wild. That is, the flows in aggregate ramp up and recover from loss in a similar manner that they would across the Internet, e.g., as determined by TCP’s response to round trip times, bottleneck bandwidths, etc.

Critical to our methodology are techniques to reproduce the application- and user-level characteristics of the flows in some original trace, e.g., session initiation according to user behavior, packet sizes according to protocol behavior, etc. We also recreate the bandwidths, latencies, and loss rates observed in the original trace. Reproducing these network conditions is important to enabling responsiveness of our generated background traffic to the characteristics of the foreground/application traffic.

Explanation	(%)	Project/Paper Title and the Conference Name
No Background Traffic	25.6	SIGCOMM '06 - Churn in distributed systems, SpeakUp, SIGCOMM '04 - Modeling P2P, Mercury, OSDI '04 - FUSE, NSDI '05- Quorum, Low bandwidth DHT routing, NSDI '04 - Macedon, Thor-SS, SIGCOMM '07 - Structured streams, NSDI '07 - SET
Constant Bit Rate Traffic	2.33	SIGCOMM '04 - CapProbe
Fixed Loss Rate	2.33	OSDI '04-FUSE
Lowered Link Capacity	2.33	NSDI '06 - DOT
Only Latencies	2.33	NSDI '06-Colyseus
"Some" Background Flows	2.33	NSDI '05 - Trickles
"Some" TCP source	2.33	SIGCOMM '04 - CapProbe
Custom Built Simulator	4.65	NSDI '05 - Myths about structured/unstructured overlays, Glacier
Pareto Flow Arrival	4.65	SIGCOMM '05 - VCP, NSDI '06 - PCP
Fixed Length Flows	2.33	NSDI '06 - PCP
Long Lived flows	4.65	SIGCOMM '05 - TFRC, SIGCOMM '07 - PERT
LRD Traffic	2.33	SIGCOMM '04 - CapProbe
Pareto Length Flows	2.33	NSDI '06 - PCP
SpecWeb	6.98	OSDI '06 - TCP offload, NSDI '06 - Connection conditioning, NaKika.
Run on PlanetLab	18.6	NSDI '06 - CoBlitz, OASIS, NaKika. NSDI '05 - Shark, Botz4Sale, NSDI '04 - Saxons, NSDI '07 - BitTyrant, SET
Real World Deployment	9.3	NSDI '06 - Overcite, NSDI '04 - BAD-FS, TotalRecall, NSDI '07 - BitTyrant
Harpoon	2.33	SIGCOMM '05 - End-to-end loss rate measurement
RON Testbed	2.33	NSDI '05 - MONET

Table 1: Literature survey of SIGCOMM, SOSP/OSDI and NSDI from 2004-2007.

Background. To motivate the importance of background traffic for a range of studies, we conducted a literature survey of SIGCOMM, SOSP/OSDI and NSDI from 2004-2007. We determined whether each paper contained a performance evaluation of a distributed system and, if so, what types of background traffic were employed in the evaluation. Overall, we found 35 papers that conducted a total of 43 such experiments. Table 1 summarizes a subset of these experiments, along with a descriptive project name and the publication venue. We divide the set of techniques into four main categories.

Our study is vulnerable to sampling bias, however we make the following high-level observations. More than 25% (11/43) of the experiments use no background traffic (NBG). The application to be evaluated is typically run on a cluster of machines with high-speed interconnect. Another 14% of the experiments account for congestion using simple models such as constant bit rate (CBR) traffic or simply constraining link latencies/capacities in a synthetic topology.

At the other extreme (bottom of Table 1), approximately 30% of the experiments employ live deployment on testbeds such as PlanetLab, RON, and Harpoon. Finally, in the middle of the table we have 25% of experiments that are done with some sophisticated models to account for background traffic, for instance, Caprobe experiments use Long Range Dependent (LRD) traffic.

Based on this study, we observe significant confusion in the community regarding whether background traffic is an important consideration in carrying out experimental evaluations. Further, there is no consensus as

to what type of background traffic should be employed. Finally, in virtually all (29/35) papers just *one* model of background traffic is used with no analysis of application sensitivity to different background traffic conditions; this can partially be attributed to the large space of possible models of traffic.

A goal of our work is to enable the community to make informed decisions about whether background traffic should be considered in a particular scenario, and if so, the particular characteristics of background traffic that are important to consider. Thus, we consider the interaction of a variety of applications with a range of competing background traffic. Ideally, we would consider all of the background traffic models summarized in Table 1 against all of the 43 experiments that we found in those papers. In this paper, we take a few steps toward this goal. For instance, we show that CBR and Poisson traffic have very similar impact on the applications we consider and that setting probabilistic loss rates does not capture the complexity of real interactions with background traffic. We also study the impact of realistic background traffic models on application performance.

Related Work. WASP [14] is perhaps most closely related to our work in spirit. It shows that HTTP performance can be significantly impacted by setting realistic delays and loss rates for the flows. The work concludes that web services cannot be evaluated on high speed LANs, but must instead consider wide-area networking effects. Relative to this effort, we consider a number of application classes and background traffic

conditions. We show that simple models of wide-area conditions (such as higher round trip times and non-zero loss rates) are insufficient to capture the effects of realistic Internet background traffic either.

Our work will benefit from ongoing work in producing realistic Internet traffic, including Tmix [9], Harpoon [19], Surge [6], and Swing [21]. We chose to generate realistic background traffic using Swing, but we expect qualitatively similar results had we employed alternative tools. We make no claims, positive or negative, about whether the traffic we generate is realistic or not. Rather, we consider a range of qualitatively and quantitatively *different* traffic conditions and show the resulting effect on application performance. However, we do like to add that before the advent of Swing, it was not possible to create realistic and responsive network traffic in a testbed environment [21]. This partially explains why researchers have been using ad-hoc traffic models in their experiments to date.

3 Methodology

We begin by describing the architecture used for carrying out the experiments followed by the list of applications we used. We then describe the background traffic models used followed by the experiments we conducted.

3.1 Architecture

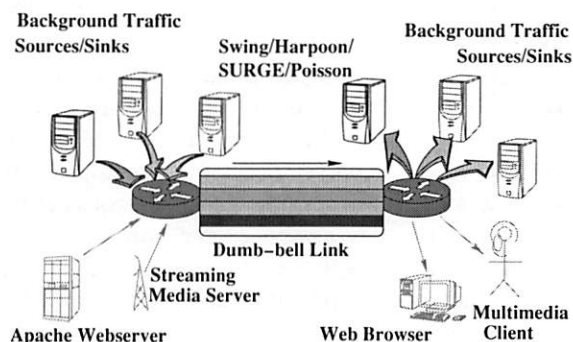


Figure 1: Evaluation architecture.

Figure 1 depicts our approach to quantifying and understanding the impact of background traffic on individual applications. We place traffic sources and sinks on either side of a constrained/dumb-bell link such that all traffic in both directions crosses the common link. Two classes of sources and sinks generate the traffic crossing the link. In the first class depicted at the bottom, nodes generate application traffic, for instance, Apache Web server and httpperf clients. Then on top, we have sources and sinks responsible for generating *background* traffic for the target link, for instance using Swing [21], Harpoon [19], SURGE [6] or simpler traffic sources such as Poisson or Constant Bit Rate (CBR). Because appli-

cation traffic and background traffic share the common link, we can quantify the impact on the application as a function of a range of background traffic characteristics.

In a real network environment, applications must compete with background traffic at multiple links between the source and destination. However, there are no known techniques to model desired background traffic characteristics at multiple successive links in some larger topology. For instance, there may be strong correlations between the background traffic characteristics of links in the topology. For the purposes of this study, we feel it reasonable to quantify and understand the impact of background traffic at a single link before attempting to extrapolate to more complex scenarios. In all likelihood, the effect of background traffic at multiple links will be even more pronounced than our findings. Hence, our results should be interpreted as a conservative estimate of the effects of background traffic, while still demonstrating application sensitivity to varying background traffic characteristics.

3.2 Applications

Of course, the impact of background traffic heavily depends upon the characteristics of the particular application under consideration. For this study, we chose three applications with diverse communication patterns and requirements: Web traffic, multimedia streaming, and end-to-end bandwidth estimation. Note that each of these applications exercise one end-to-end path, and hence, we explicitly omit more distributed applications such as BitTorrent that simultaneously exercise multiple independent Internet paths. While this class of application is important, considering complex topologies is beyond the scope of this paper (see above). We did run experiments (not discussed further here) for the case where all BitTorrent clients were subject to a dumbbell topology; these results were qualitatively similar to our findings for HTTP.

Web Traffic For Web applications, we set out to determine the effect of background traffic on the response times perceived by end clients. We placed a single Apache Web server on one side of the dumbbell (e.g., the bottom left in Figure 1). We programmed httpperf clients to fetch objects of various sizes from the server and placed them on the other side of the dumbbell. The links connecting the clients and server to the dumbbell have large capacity and low latency, such that the dumbbell is the constrained link for all client-server communication (we vary the capacity of the dumbbell link in various experiments described below).

To generate background traffic, we place sources and sinks of the appropriate traffic generator on either sides of the target link (top left/right in Figure 1). We set the bandwidths, latencies, and loss rates of the links connect-

Trace ↓	Secs	Trace BW		Trace Collection Date	Number of flows	Dominant applications	Unique IPs (1000s)
		Aggregate (Mbps)	Dir0 (Mbps)				
Auck	600	5.5	3.3	June 11, 2001	155 K	HTTP, SQUID	3
Mawi	900	17.8	7.8	September 23, 2004	476 K	HTTP, RSYNC	15
Mawi2	900	11.9	10.8	December 30, 2003	160 K	HTTP, NNTP	8

Table 2: Trace characteristics for three different links.

ing background traffic sources and sinks based on the traffic generation model. For instance, we simply play back CBR traffic over unconstrained links; whereas for Swing, we assign latencies, bandwidths and loss rates based on observed path characteristics in some original packet trace [21].

Multimedia Traffic The second application class we consider is video streaming. Video clients are sensitive to the arrival times of individual packets, whereas web clients are typically sensitive to end-to-end transfer times. Overall, we wish to quantify the impact of various types of background traffic on client-perceived video quality. For streaming audio/video we use the free version of Helix on the server side and Real Player on the client side. We generate background traffic across the dumbbell topology as with Web traffic.

Bandwidth Estimation Tool We chose bandwidth estimation for our third application. While not an end application, it displays fundamentally different characteristics than our first two applications and is a building block for many higher-level services. We employ Pathload [10], and pathChirp [17] tools for our study. We place bandwidth senders and receivers along with competing traffic generators across the dumbbell topology identically to our configuration for Web and video streaming.

3.3 Traffic Generation

We consider four techniques for generating competing background traffic, in increasing order of complexity and realism. First, for constant bit rate (CBR) traffic, we wrote simple sources to generate packets at a specified rate to sinks on the opposite side of the dumbbell link. In aggregate, the sources generate a target overall rate of background traffic. Second, for Poisson traffic, we modify the sources to generate traffic with byte arrival per unit time governed by a Poisson process with a given mean. We evaluated variants of CBR and Poisson using both UDP and TCP transports.

While CBR and Poisson processes do not capture the complexities of real Internet traffic [16], we wish to quantify the resulting differences in end-to-end application behavior relative to more realistic, but complex traffic generation techniques. Hence, for our third technique, we modify the sources and sinks to play back packets in

the exact pattern specified by an available tcpdump of traffic across an existing Internet link (Table 2). One drawback of this approach is that the generated background traffic is not congestion responsive. That is, the traffic will be played back in exactly the same pattern as the original trace irrespective of the behavior of the application traffic. Another drawback is that it is difficult to extrapolate to alternative, but similar, scenarios when playing back a fixed trace (e.g., changing the available bandwidth across the constrained link, the distribution of round trip times between sources and sinks, etc.).

Thus, for our fourth technique, we use Swing [21] to generate responsive and realistic network traffic. Swing is a closed-loop, network responsive traffic generator that qualitatively captures the packet interactions of a range of applications using a simple structural model. Starting from a packet trace, Swing automatically extracts distributions for user, application, and network behavior. It then generates live packet traffic corresponding to the underlying models in a network emulation [15, 20] environment running commodity network protocol stacks. Because Swing generates the traffic using real TCP/UDP stacks, the resulting traffic is responsive both to foreground traffic and varying characteristics of the constrained link and end-to-end loss rates/round trip times.

Swing extracts a range of distributions from an original trace to model user behavior, e.g., think time between requests, application behavior, e.g., distribution of request and response sizes, and network characteristics. One particularly important aspect of network traffic that Swing is able to reproduce is burstiness in the packet arrival process at a range of time scales [21]. Doing so requires Swing to assign latencies, bandwidths, and loss rates to the links leading to and from the shared link in our evaluation architecture based on the observed distribution of round trip times, link capacities, and loss rates in some original trace. This way, TCP flows ramping up to their fair share bandwidth or recovering from loss will do so with statistically similar patterns (including burstiness in aggregate) as in the original trace conditions.

An additional benefit of employing high-level models of user, application, and network characteristics for background traffic is that it becomes possible to modify certain model parameters to extrapolate to alternative scenarios [21]. For instance, when reducing the round

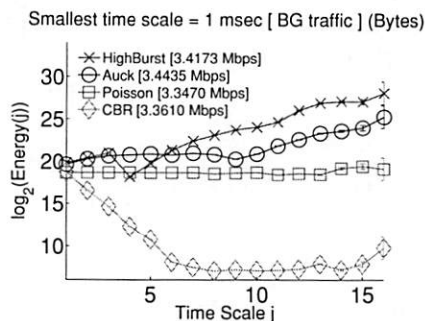


Figure 2: Auck-based background traffic (Energy plot).

trip times for flows, overall burstiness tends to increase because flows tend to increase their transmission rates more quickly. We refer to alternate background traffic generated by perturbing distributions of various parameters in Swing as *variants* of the original trace.

We run all our experiments, including the sources and sinks for both foreground (Web, multimedia, bandwidth estimation) and background traffic (CBR, Poisson, Playback, Swing) in the ModelNet emulation environment [20]. For our experiments, we configure all sources and sinks of both foreground and background traffic to route all of their packets through ModelNet. Briefly, ModelNet subjects each packet to the per hop bandwidth, delay, queueing, and loss characteristics of the target topology by inspecting both the source and destination of the packet and routing it through the emulated topology. ModelNet operates in real time, meaning that it moves packet from queue to queue in the target topology before forwarding it on to the destination machine assuming that the packet was not dropped. Earlier work [20] validates ModelNet's accuracy using a single traffic shaper at traffic rates up to 1Gbps (we can operate at higher speeds by employing multiple traffic shapers in parallel).

3.4 Topology and Experiments

We run our experiments on a cluster of commodity workstations, multiplexing multiple instances on each workstation depending on the requirements. For the experiments in this paper, we use eight 2.8 Ghz Xeon processors running Linux 2.6.10 (Fedora Core 2) with 1GB memory and integrated Gigabit NICs. We generate background traffic using 1000 nodes in the emulated topology (meaning that we multiplex hundreds of emulated nodes onto each physical machine).

For example, for a 200Mbps trace, assuming an even split between generators and listeners (four machines each), each generator would be responsible for accurately initiating flows corresponding to 50Mbps on average. Each machine can comfortably handle the average case, though there are significant bursts that make it important to "over-provision" the experimental infras-

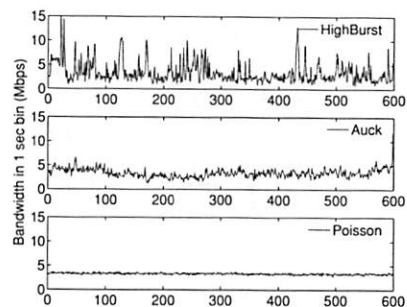


Figure 3: Auck-based background traffic (Time Series).

tructure. To avoid contention between physical resources with the traffic generator we use a separate set of machines to host the target application; for instance, httpperf clients. In this fashion, any performance impact and variations that we measure results purely from network behavior.

We first describe the traces we use to generate realistic background traffic for the experiments in this paper. We use traces from Mawi [12], a trans-Pacific line, as well as an OC3c ATM link traces from the University of Auckland, New Zealand [5]. These traces come from different geographical locations (New Zealand, Japan) and demonstrate variation in application mix, average throughput, number of users etc. (see Table 2). All traces were taken on 100Mbps links. The Mawi traces were constrained to 18Mbps over long time periods (though it could burst higher).

While all original and the corresponding Swing-generated traces are bidirectional, we focus on the impact of competing traffic in one direction of traffic (Dir0 in Table 2) for simplicity in our plotted results. In other words, we design all experiments such that the dominant direction of application traffic (HTTP responses, video, bandwidth estimation packet train) matches Dir0 of the generated background traffic.

We use wavelet-scaling plots [1, 8, 22] to characterize traffic burstiness. Intuitively, these plots allow visual inspection of burstiness for a range of timescales. The x-axis of these plots shows the time scale on a log scale and the y-axis shows the corresponding energy value. Higher levels of energy correspond to more burstiness. Figure 2 plots burstiness corresponding to five variants of the Auck trace for a 20Mbps link (along with the average bandwidth for each variant in square brackets). We configured Swing to generate constant-bitrate (CBR) and Poisson traffic with the same average bandwidth as the Auckland (Auck) trace. In addition to reproducing the Auck trace using Swing, we also created a very bursty traffic variant, called *HighBurst* (HB), by setting the round trip times artificially to 4ms while generating traffic using Swing. The lower round trip times (relative to

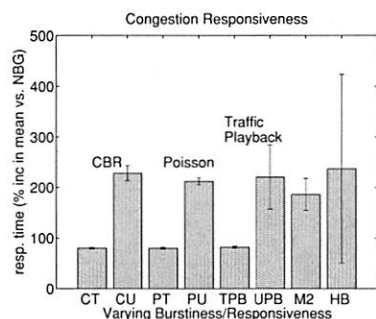


Figure 4: “Simple” models are inaccurate.

the distribution in the original trace) means that TCP traffic ramps up more quickly, recovers more quickly from losses, etc. For an alternate visualization of the relative difference in burstiness, consider the time series plots for the same traffic models shown in Figure 3. The Poisson variant generates a relatively fixed bandwidth whereas HighBurst variant peaks to 15Mbps at times, compared to Auck. Note that while the average bandwidth consumption for all traces are comparable, finer-grained behavior varies significantly. Further, none of the variants come close to congesting a 20 Mbps link.

4 Results

We subject each of our three target application classes to the various types of background traffic described above with the goal of answering the following questions:

- i) What aspects of “realism” should we reproduce? Is it sufficient to simply “replay” individual packets in some measured trace or does the generated background traffic need to be TCP responsive and react appropriately based on the end-to-end network characteristics of the traffic sources and sinks? (§ 4.1)
- ii) Can probabilistic packet drops substitute for real competing background traffic? (§ 4.2)
- iii) Does burstiness of background traffic matter or is it sufficient to reproduce average bandwidth? (§ 4.2, 4.4)
- iv) Are some applications more sensitive to background traffic than others? (§ 4.3)
- v) Is application behavior sensitive to slight variations in the background traffic characteristics? (§ 4.4)

4.1 Background Traffic Responsiveness

The first question we consider is the importance of realistically playing back background traffic characteristics. We consider three techniques for doing so: i) scheduling per-packet transmissions using UDP connections to match the exact timings (at 1 ms granularity) and packet arrival processes found in some original trace; ii) scheduling per-packet transmissions using a single TCP connection to attempt to match the packet arrival process found in the original trace; and iii) extracting user, application, and network characteristics from the original

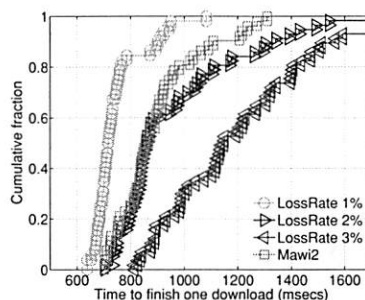


Figure 5: Deterministic loss rates never suffice.

trace and playing back TCP flows whose communication patterns are statistically similar to the original without making an attempt to match the patterns found in the original trace.

The first technique (UDP) is not responsive to the characteristics of foreground traffic. Thus, it will not back off in the face of congestion. The second technique (TCP) is responsive, but, unfortunately it becomes impossible to perform precise packet scheduling for TCP connections. Further, because we have no knowledge of the end-to-end network characteristics for the TCP flows we play back, it is not possible to verify that the response to congestion would match the behavior of flows from the original trace (e.g., because of variations in round trip times or losses elsewhere in the network). Finally, because it employs a single connection to multiplex the behavior of a much larger number of flows, its behavior is unpredictable. The third technique, corresponding to Swing-based [21] playback, promises to be the most faithful but is also the most complex and requires more resources (i.e., logic to source and sink traffic from individual hosts) for trace playback.

To establish the accuracy for each of these techniques, we run httpperf clients requesting 1 MB files from an Apache Web server sharing the bottleneck link with the Mawi2 trace. We set the shared link (the point of contention between httpperf and background traffic) to 15 Mbps. We choose 15 Mbps to ensure that the background traffic attempts to consume a significant portion of available resources (§ 4.2 onwards relaxes this assumption). We are interested in understanding the response time for HTTP as a function of the characteristics of the background traffic. As described earlier, the links connecting the httpperf/Apache nodes to the shared link are unconstrained (large capacity and low latency), so that traffic shaping takes place only at the target link. During each experiment we fetch files back-to-back, using a single client-server pair for 10 minutes.

Figure 4 shows the results for different classes of background traffic. For each scenario, we plot the mean and standard deviation of response time increase relative to the NBG (No Background Traffic) case. Background

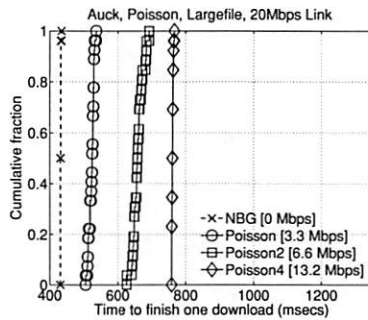


Figure 6: Poisson background traffic.

traffic consumes 66% of the shared link's capacity in all cases. The first two bars plot slowdown for CBR traffic with both TCP and UDP playback (CT and CU) configured to consume as much average bandwidth as the Mawi2 trace. The next two bars show slowdown for TCP and UDP variants of Poisson (PT and PU). For both of the models (CBR and Poisson) UDP variants impacted responses time much more (200% increase vs 100% increase) than TCP variants exposing the limitation of UDP based traffic generators. Because UDP sources are not congestion responsive, they have a much larger impact on HTTP. The next two bars show the slowdown for TCP- and UDP-based playback (TPB/UPB) of Mawi2. TPB has much less impact than Swing-based playback of Mawi2 (M2), likely because its use of a single TCP connection means that the generated background traffic is less bursty. Finally, UPB results in larger slowdown than Swing-based playback because it is not congestion responsive.

Given the above results, we conclude that *simple techniques for "playing back" background traffic, such as UBP and TBP, may result in significant inaccuracy as the aggregate traffic across a link approaches the link's capacity*. Thus, for the remainder of this paper, we employ Swing to play background traffic corresponding to some original network condition and compare it to other variants such as CBR/Poisson.

Another popular technique in the literature for capturing the complexity of real background traffic is to set loss rates for particular links, with the goal of capturing the effects of losses caused by competing traffic. To determine whether this technique could capture the effects of more complex traffic scenarios, we next measure the performance of httpperf when setting various loss rates for the shared link. In all other respects, this experiment is identical to the case where we run with no background traffic. Figure 5 shows that httpperf's behavior when crossing a link with a range of loss rates differs from its behavior when competing with realistic Mawi2 traffic, again, across a 15Mbps bottleneck link. For instance, with losses of 1% the CDF of retrieval times is too far to the left of Mawi2. On the other hand, higher

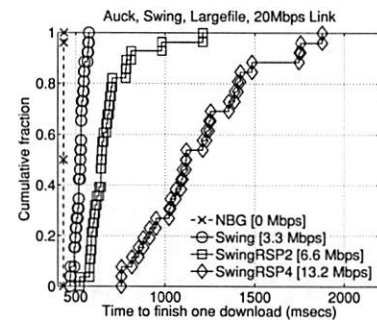


Figure 7: Swing background traffic.

loss rate settings, while shifting the CDF to the right, increase the size of the tail too. We hypothesize that the *difference in behavior results from the independent nature of losses when setting any fixed loss rate (relative to the bursty losses common for Internet links) and the fact that setting loss rate alone does not capture any of the effects of increased queueing delay (and hence increased round trip times) leading up to the point of loss*.

4.2 Httpperf/Apache

Having established the appropriate technique for transmitting background traffic, we now turn our attention to understanding the impact of background traffic on HTTP transfers. In order to arrive at a conservative estimate of the impact of background traffic we first experiment with the Auck trace (lowest bandwidth and least bursty). We begin by examining the impact of varying bandwidths of background traffic (based on Auck) on httpperf performance. We fetch 1 MB files across a 20Mbps link and vary the load placed by background traffic by generating traffic for three different average throughputs (3.3, 6.6, and 13.2 Mbps) corresponding to variants of the Auck trace. In the baseline (3.3 Mbps) case, we employ Swing parameterized by the original Auck trace. In the alternative cases, we modify the distribution of response sizes such that the average bandwidth increases by a factor of 2 and 4 (traces SwingRSP2/SwingRSP4), resulting in average bandwidths of 6.6 Mbps and 13.2 Mbps respectively. We compare the impact of Swing-generated traffic to those of TCP-generated CBR and Poisson traffic of the same average bandwidth.

Figures 6 and 7 show the CDFs of download times corresponding to various bandwidth/burstiness combinations for background traffic. Along with the legend name we show in square brackets the average bandwidth of the background traffic for reference. The effects of CBR and Poisson traffic are similar, so we only plot the results for Poisson (relative to Poisson, the CBR curves are virtually vertical with the same median value). As shown in Figure 6, the impact of Poisson (and hence CBR) traffic is almost entirely predicted by the average level of back-

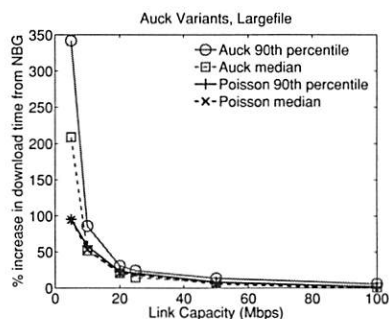


Figure 8: Varying link capacities.

ground traffic bandwidth. The distribution of download times shifts to the right, but with little variation in response times as the average level of background traffic increases. The Swing-generated Auck trace has a more varied impact on application performance (Figure 7). With low levels of competing traffic (3.3 Mbps), the distribution of download times is similar to Poisson. Also of interest is the fact that the performance for the 5th-percentile of retrievals is actually faster for Auck than for Poisson traffic for both the 6.6 Mbps (SwingRSP2 vs. Poisson2) and the 13.2 Mbps (SwingRSP4 vs. Poisson4) bandwidths. In these cases, the flows were lucky to be subject to less competing traffic than their counterparts in the Poisson trace. Bursty traffic means that periods of high activity coexist with periods of lower activity. Moving to higher levels of average utilization, the curves become significantly skewed. For instance, the 90th percentile of download time for the Auck (AuckRSP4) trace at 13.2 Mbps is 1484 ms compared to 759 ms for Poisson (Poisson4) traffic.

Thus, less bursty background traffic means that performance is governed by the average amount of available bandwidth and, expectedly, there is relatively small variation in download times across individual object retrievals. When background traffic is steady, HTTP performance is predictable. As burstiness increases, the mean download time increases, as does the variation in performance. Some flows can get lucky, behaving almost as if there is no background traffic; while others may be unlucky with significantly worse performance than the mean.

Next, we consider the sensitivity of HTTP performance to background traffic characteristics as a function of the fraction of shared link capacity occupied by the background traffic. That is, it could be the case that background traffic characteristics (e.g., burstiness) are only important when consuming a significant fraction of link capacity. Figure 8 shows the impact on download times for varying levels of background traffic with average bandwidths of 3.3Mbps (same as the Auck trace). The y-axis shows the slowdown (median as well as 90th-percentile) of HTTP transfers of large 1 MB files relative

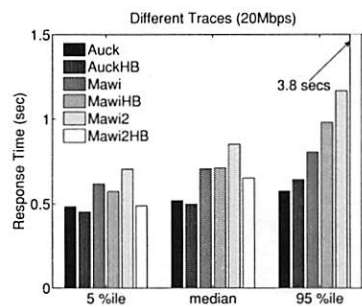


Figure 9: Simply generating “bursty” traffic is not enough.

to the NBG case as a function of the capacity (5-100 Mbps) of the shared link on the x-axis. On the left side of the graph we have cases corresponding to a highly utilized link; while on the right side, the background traffic consumes a small fraction of overall capacity. For large files and low levels of link utilization, the graphs show that burstiness of background traffic does not matter. For instance, for a 50Mbps link the impact on median response time is independent of the burstiness; the slowdown largely corresponds to the fraction of the link consumed by the background traffic.

When background traffic consumes a significant portion of link capacity however, it is sufficient to cause significant losses for the HTTP transfers. Thus, at the 90th percentile of slowdown, we find that transfers competing with bursty traffic completed significantly more slowly, around a factor of 1.5 for Auck, than with a less bursty traffic source, i.e., Poisson. However, unlike median response times, this relative ordering is present even when overall capacity is high (e.g., 100 Mbps). *Thus, for large files, burstiness of traffic matters at all levels of link utilization, but more so at high levels of utilization.*

The impact on transfer times for different file sizes as a function of different levels of burstiness of background traffic is further explored in § 5.2. *Overall we conclude that impact on download time for web transfers is a function of the size of the download and the average bandwidth of background traffic as well as its burstiness.*

Finally, we consider whether it is sufficient to simply playback a “bursty” traffic source or whether traffic sources with different burstiness characteristics will have different impacts on HTTP performance. Thus, we considered the impact of six different bursty background traffic characteristics competing for a shared 20 Mbps link. We considered background traffic corresponding to Auck, Mawi, and Mawi2. We further modified each of these sources to be high burst variants (“HB”) by setting the round trip times for all flows to 4msec while generating traffic using Swing. We plot the slowdown of HTTP transfers (1MB file) relative to the NBG case at the 5th, median, and 95th percentile in Figure 9.

There are a number of interesting results here. First,

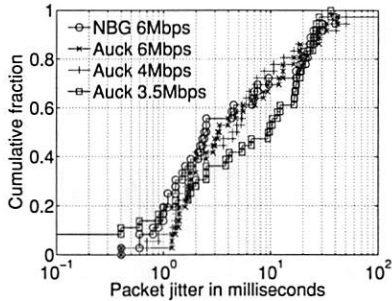


Figure 10: Varying capacity.

the Mawi2 trace appears to have a larger impact on HTTP than Mawi, even though it consumes significantly less aggregate bandwidth (see § 5 for explanation). Further, the HB-variant of Mawi2 has significantly less impact on HTTP performance at the 5th and 50th percentile, but more than a factor of 5 more impact at the 95th percentile. Its highly bursty nature means that a significant number of flows are lucky and suffer comparable slowdown to the less bursty cases. However, a number of flows are extremely unlucky and suffer large slowdowns. The tail for the Mawi2HB case is significantly longer as well, an undesirable character for HTTP transfers where the humans in the loop typically value predictable behavior. *This experiment shows that although burstiness of traffic is important to consider, simply reproducing “some” bursty traffic is not enough.*

4.3 Multimedia

We next consider the impact of background traffic on multimedia traffic. We aim to understand the impact as a function of the capacity of the link, the burstiness of the generated traffic as well as the amount of client-buffering. We run the publicly available Helix Server to serve real media content to RealPlayer. As with all our experiments, the application traffic competes with various types of background traffic at the shared link. Like `httperf`, we consider 1 client-server pair. We tested with various streaming rates but the results in this paper are for a 450Kbps CBR stream encoded using RealVideo codec at 30 frames per second that runs for 62 seconds.

One important question for multimedia traffic is the “metric of goodness” for a multimedia stream, which should correspond to the quality of the video played back. However, developing such a metric based on the data stream received at clients is challenging. Thus, as a proxy for such a quality metric we use a range of statistics based on the stream delivered to the client and corroborate it with visual inspections. For a 450 kbps video stream, we can roughly assume that receiving more than 450 kbps of instantaneous bandwidth results in acceptable video quality at that particular point in time.

Another important metric of interest is *jitter*, the time

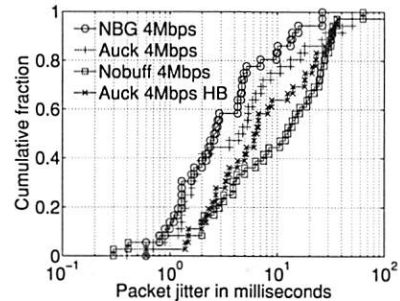


Figure 11: Varying burstiness and client-buffering.

between successive packets received by the client. Lower levels of jitter will correspond to higher video quality. For our first experiment, we run with Auck background traffic (3.3 Mbps average) and compare its impact on jitter for shared links with 3.5 Mbps, 4 Mbps, and 6 Mbps capacity (Realplayer was insensitive to background traffic at higher levels of link capacity). Figure 10 shows the distribution of inter-packet timing (jitter) for this experiment. We also perform an experiment in the absence of any background traffic (NBG) for baseline. For a 6 Mbps target link, background traffic does not change the quality of video (inspected by viewing the video) whereas for a 4 Mbps link the degradation is moderate. It is only when we constrain the link capacity to 3.5 Mbps that video quality suffers significantly. *One conclusion is that unless we are operating in extreme scenarios (available bandwidth approximately equal to the bandwidth of the stream), RealPlayer is relatively insensitive to bursty background traffic.*

To test this hypothesis, we next attempt to stress the limits of RealPlayer. We modify the client-side implementation to reduce the amount of buffering from a default of 10 seconds to 0 seconds. Figure 11 plots the distribution of jitter with and without buffering for the 4Mbps link. With 10 seconds of buffering, background traffic had no impact on jitter (Auck 4Mbps). However, there is significant impact (verified by visual inspection) when we remove the buffering. Without buffering, the real server attempts to retransmit lost packets more aggressively in order to meet real time deadlines, consuming more network resources and negatively impacting overall jitter. With sufficient buffering, the server can afford to be more relaxed about retransmissions, resulting in an overall smoother transmission rate.

Finally, we consider the highburst background traffic source by setting RTTs in Swing to 4msec. In this case, flows ramp up and down very quickly causing bursty traffic on the shared link. Figure 11 also plots the distribution of jitter corresponding to this experiment. The result shows that even for tight links and bursty background traffic the performance degradation in realplayer is moderate (again verified by visual inspection).

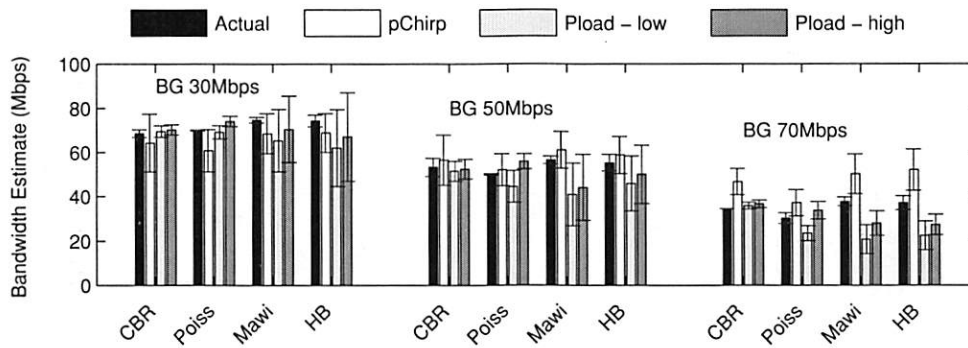


Figure 12: Augmenting to and validating pathload vs. pathchirp experiments from [17].

Overall, we conclude that for RealPlayer, unless we are operating in regimes of low buffering, low available bandwidth or very high burstiness, any reasonable background traffic model should suffice in performance evaluations for systems similar to RealPlayer; in most cases, RealPlayer's default buffering will likely result in acceptable quality of service for a range of background traffic characteristics, as long as sufficient average bandwidth levels are available. Interestingly, RealPlayer's (successful) buffering and retransmission scheme is informed by significant experience with live Internet deployments subject to a range of bursty competing cross traffic.

4.4 Bandwidth Estimation

For our final experiments, we consider the sensitivity of bandwidth estimation tools to background traffic characteristics. We use Pathload [10] and pathChirp [17] for our study because they are publicly available and because recent studies indicate that they are among the most accurate for bandwidth estimation [18]. To determine the sensitivity of existing tools to a range of background traffic characteristics, we repeat experiments from earlier published work comparing the relative merits of Pathload to pathChirp [17]. In these experiments, the authors employed Poisson and CBR models for background traffic. Below, we show that at least the conclusions from this earlier work may be reversed if the experiments had considered more realistic background traffic characteristics.

For our experiments, we overprovision all links such that the available bandwidth measured for the path is determined by the bandwidth available on the shared link in our model topology. We set Pathload's timeout to five minutes, and we report the average and standard deviation for the *low* and *high* estimates of available bandwidth across 25 runs. In practice, a small spread between the low and high estimates of available bandwidth and a low standard deviations for multiple reported values reflects likely accurate bandwidth estimates. PathChirp, on the other hand, periodically outputs an estimate of the

available bandwidth and does not distinguish between a low and a high value.

Given our exact knowledge of the generated background traffic, we also calculate the true values of the available bandwidth for one second bins. In the graphs that follow, we plot the percentage error reported by each bandwidth estimation tool relative to our calculated values for available bandwidth.

The experiments from [17] compare available bandwidth across a 100Mbps link as measured by pathload and pathchirp against various flavors of background traffic. They employ CBR UDP traffic and UDP Poisson traffic to create three different scenarios with available bandwidths of 30, 50 and 70 Mbps. Figure 12 shows these results. Additionally, we also playback Mawi, and a HighBurst variant of Mawi (labeled HB) for the three levels of available bandwidth. In each case, we increase the number of user-sessions (in Swing) by an appropriate value to match the levels of bandwidth consumed by CBR and Poisson traffic. For instance, to get the available bandwidth of 70Mbps, we multiplied the number of sessions for Mawi by 3.3 times.

We initially discuss the results for Poisson and CBR models, reproducing the earlier experiments [17]. First, we confirm the earlier result [17] that when background traffic is low (BG 30Mbps) or moderate (BG 50Mbps) the estimates of pathload as well as pathchirp are close to the actual values. We also observe that indeed pathchirp takes 10-20% the amount of bytes consumed by pathload to arrive at similar estimates (not shown here). However, we note that a couple of results differed. For instance, pathchirp overestimated the bandwidth for a heavily utilized link, i.e. when background traffic was around 70 Mbps. Similarly, for low link utilization (BG 30Mbps), pathchirp is slightly less accurate than pathload.

We next move to the effects of bursty background traffic, not considered by the earlier work, the bars corresponding to Mawi and HB in Figure 12. We make a number of new interesting observations here. First, the results for both tools degrade when competing with more bursty

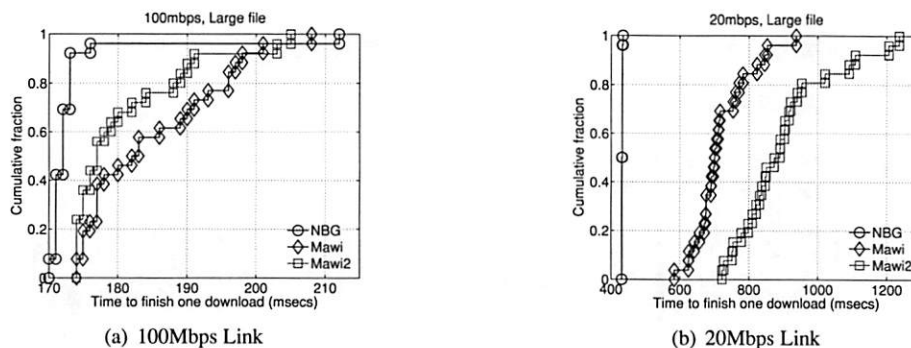


Figure 13: Significance of traffic in the reverse direction.

traffic sources, indicated by either an inaccuracy in the mean or large standard deviation. PathChirp appears to be a bit less sensitive to burstiness of background traffic. We also no longer see that pathchirp is always conservative in its estimates. In fact, when the background traffic occupies a significant portion of the link we see that pathchirp overestimates available bandwidth, e.g., Mawi/HB traffic with 70 Mbps background traffic. The authors [17] saw that both pathchirp and pathload estimates are close to each other, however, we see that there are times when the estimates are very different from each other; for instance, Mawi for 50 Mbps.

The conclusion from these results is that it is difficult to predict a priori which bandwidth estimation tool gives better results and that realistic network traffic characteristics can impact study results. System behavior is a function of the average amount of background traffic as well as burstiness. For instance, if we hypothesize that only background traffic's average throughput is important, then we can disprove it by observing the case for 30 Mbps BG traffic. For CBR background traffic, pathload is the more accurate tool, whereas for HB, pathchirp is more accurate in its estimate. Similarly if we hypothesize that only burstiness matters, then we can disprove that by looking at HB traffic. At 30Mbps we would prefer pathchirp for HB traffic whereas at 70Mbps pathload shows superior accuracy. We leave the task of attributing the sensitivity of these tools to the underlying algorithms to orthogonal future work.

5 Case Studies

Considering our evaluation to this point, sensitivity to background traffic characteristics is application specific. Certain applications, such as bandwidth estimation are highly sensitive to background traffic while others, such as RealPlayer, are relatively insensitive except in extreme cases. We now turn our attention to some additional important characteristics of background traffic and their impact on end-to-end applications.

5.1 Bi-directional Traffic Characteristics

To this point, we have largely focused on traffic characteristics in one direction of a link (though in all cases, we played back bi-directional traces). We now consider a case where traffic characteristics in the reverse direction can impact application performance depending on the deployment setting. In the first experiment, we repeat retrievals of 1MB files using httpperf/Apache (as Section 4.2) across a shared 100 Mbps target link. We use the background traffic models corresponding to the two Mawi traces in Table 2. The traces are in increasing order of bandwidth in the direction of flow of HTTP responses (Dir 0). One would expect that the response time distribution would correspond to the relative bandwidth of each of these traces.

Figure 13a) plots the CDFs of retrieval times for this experiment. Mawi2 is of higher bandwidth than Mawi (Table 2) but the CDF is to the left of Mawi as a result of background traffic in the reverse path. This effect is more pronounced for Mawi as it has 10 Mbps of traffic in the reverse direction versus 1 Mbps for Mawi2. This relative ordering, however, is not present when we repeat the experiments with a 20 Mbps shared link as shown in Figure 13b). At 20 Mbps, the forward direction traffic dominates for both Mawi and Mawi2 so the effects of congestion on the reverse path is less pronounced. Thus, one simple conclusion is that *background traffic in the reverse direction can impact application performance though the dominant direction is difficult to predict a priori.*

5.2 Burstiness at Various Timescales

We have shown that background traffic with the same average bandwidth, but differing burstiness characteristics will have varying impact on application behavior. We now consider the question of whether burstiness at particular timescales (for instance, burstiness at millisecond versus second granularity) has differing impact on application performance. Generating traffic that selectively and precisely varies burstiness at arbitrary timescales is an open problem. However, we can alter burstiness in

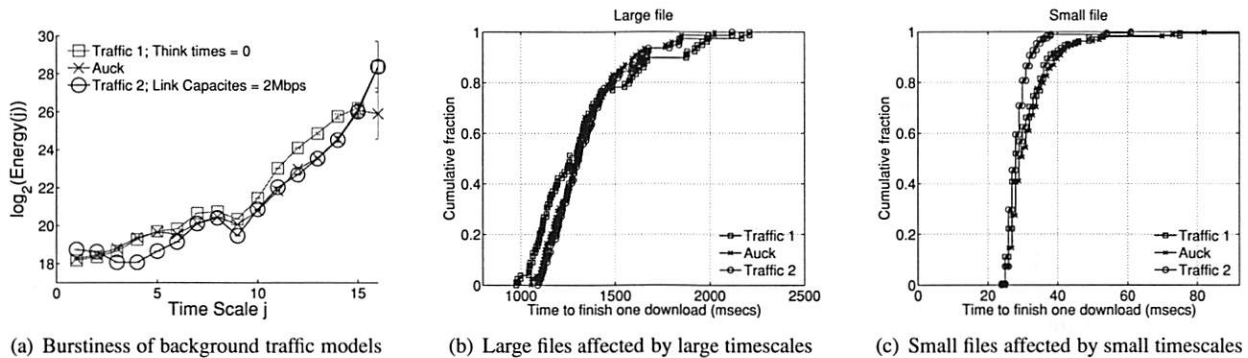


Figure 14: Varying burstiness at small and large timescales to understand impact on file download times.

a coarse manner at relatively small and large timescales. To vary burstiness at large timescales, we set Swing's user think time distribution to 0 for all requests; meaning, for instance, that for each user session, requests for subsequent objects will be initiated immediately after the previous request completes. Figure 14a) shows the resulting energy plot. Note that traffic becomes more bursty at timescales 11 – 14 (corresponding to the 1-8 sec range) relative to the default Auck trace. To vary burstiness at timescales 3 – 6 (4-32 ms), we restrict all links in the emulated topology for Swing sources to 2 Mbps. Figure 14a) also shows the energy plot for this case. Let us call these two new traces *Traffic1* and *Traffic2*. We will next try to understand if these seemingly small differences in burstiness impacts application performance.

We run HTTP experiments across a shared 10 Mbps link. The results in Figure 14b) shows the effect of varying burstiness on a relatively large 1 MB file transfer. We see that varying burstiness at small time scales (*Traffic2*) has relatively little impact on the distribution of response times. Burstiness at larger time scales (*Traffic1*) skews the CDF around the median. Few flows finish early and few take longer compared to Auck/*Traffic2*. We hypothesize that over a long transfer, burstiness at small time scales (milliseconds) averages out over the lifetime of the connection. Burstiness over multiple seconds however will more significantly impact transfers that complete in just over one second by default.

The situation reverses itself when we consider the distribution of download times for the same experiment but with 4 KB objects as shown in Figure 14c). In this case, increased burstiness at large timescales (*Traffic1*) has no impact on the distribution of performance relative to the baseline. However, the decreased burstiness at small timescales for *Traffic2* relative to Auck results in improved download times, especially above the 80th percentile. Note that *Traffic2* displays reduced burstiness in the 4-32 ms timescales (Figure 14a), and that retrieving a 4 KB takes 29 ms in the median case for Auck. From this initial experiment, we hypothesize that *for a given*

level of background traffic, its burstiness characteristics at timescales corresponding to the duration of individual application operations will have the most significant impact on overall application behavior.

6 Discussion

While this paper shows the importance of subjecting network services to a range of background traffic conditions, there are a number of remaining open questions. First is the need for a suite of background traffic that capture the range of conditions likely to be experienced during live deployment. We have shown that the bursty traffic present on the Internet can dramatically affect application behavior relative to synthetic traffic models. However, we cannot yet characterize the full range of network conditions likely to be present on the Internet.

Next, we have not yet shown the best way to generate the background traffic. We have shown one plausible technique employing the Swing traffic generator. However, Swing requires multiple machines to generate the traffic and a network emulator to appropriately shape individual flows. And yet, our current understanding indicates that recreating Internet traffic burstiness critically depends on recreating appropriate network characteristics and closed-loop responsive traffic sources and sinks that, for example, obey the dynamics of TCP [21].

Finally, our analysis considers the effects of background traffic on a single link between sources and destinations. Under realistic deployment scenarios, application traffic must interact with background traffic at multiple links across the network. Our initial experiments, not shown for brevity, indicate that the impact of bursty background traffic is even more pronounced and unpredictable when considering more complex network topologies. We believe that capturing this full complexity will be challenging in the short term, but it would be valuable to determine whether relatively simple models can account for most of the additional impact that comes from more complex topologies.

7 Conclusion

We set out to answer a simple question: When running simulation or emulation experiments, what kind of background traffic models should be employed? Additional motivation comes from recent interest in accurately recreating realistic background traffic characteristics. While there have been significant advancements in this space, there is relatively little understanding of what aspects of background traffic actually impact application behavior. To fill this gap, we quantified the interaction of applications with a variety of background traffic models. We found that, for instance, HTTP is sensitive to the burstiness of background traffic depending on the dominant size of transferred objects; multimedia applications have been engineered to be relatively insensitive to traffic burstiness; and bandwidth estimation tools are highly sensitive to bursty traffic because unstable link characteristics make convergence to stable estimates difficult. We also observed that characteristics of background traffic in both directions of a link can impact application performance. Finally, we hypothesize that each application is sensitive to burstiness of traffic at particular application-dependent timescales.

Acknowledgements

We would like to thank the reviewers for their insightful and detailed comments that helped improve both the quality and presentation of this paper. Ryan Braud, Chris Edwards and Marvin McNett helped set up the infrastructure behind our experiments.

References

- [1] ABRY, P., AND VEITCH, D. Wavelet Analysis of Long-range-dependent Traffic. *IEEE Transactions on Information Theory* (1998).
- [2] ALAN SHIEH AND ANDREW C. MYERS AND EMIN GUN SIRER. Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility. In *NSDI* (2005).
- [3] ANDERSON, T., COLLINS, A., KRISHNAMURTHY, A., AND ZAHORJAN, J. PCP: Efficient Endpoint Congestion Control. In *NSDI* (2006).
- [4] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIERES, D. Shark: Scaling File Servers via Cooperative Caching. In *NSDI* (2005).
- [5] Auckland-VII Trace Archive, University of Auckland, New Zealand. <http://pma.nlanr.net/Traces/long/auck7.html>.
- [6] BARFORD, P., AND CROVELLA, M. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *MMCS* (1998).
- [7] CHEN, J., GUPTA, D., VISHWANATH, K. V., SNOEREN, A. C., AND VAHDAT, A. Routing in an Internet-Scale Network Emulator. In *Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)* (October 2004).
- [8] FELDMANN, A., GILBERT, A. C., HUANG, P., AND WILLINGER, W. Dynamics of IP Traffic: A Study of the Role of Variability and the Impact of Control. In *ACM SIGCOMM* (1999).
- [9] HERNANDEZ-CAMPOS, F., SMITH, F. D., AND JEFFAY, K. Generating Realistic TCP Workloads. In *CMG2004 Conference* (2004).
- [10] JAIN, M., AND DOVROLIS, C. Pathload: A Measurement Tool for End-to-end Available Bandwidth. In *PAM* (2002).
- [11] MAHADEVAN, P., HUBBLE, C., HUFFAKER, B., KRIKOUKOV, D., AND VAHDAT, A. Orbis: Rescaling Degree Correlations to Generate Annotated Internet Topologies. In *ACM SIGCOMM* (August 2007).
- [12] MAWI Working Group Traffic Archive. <http://tracer.csl.sony.co.jp/mawi/>.
- [13] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *OSDI* (2004).
- [14] NAHUM, E. M., ROSU, M.-C., SESHAN, S., AND ALMEIDA, J. The Effects of Wide-area Conditions on WWW Server Performance. In *SIGMETRICS/Performance* (2001).
- [15] OF UTAH, U. Emulab.Net: The Utah Network Testbed. <http://www.emulab.net>.
- [16] PAXSON, V., AND FLOYD, S. Wide-Area Traffic: The Failure of Poisson Modeling. In *IEEE/ACM Transactions on Networking* (1995).
- [17] RIBEIRO, V., RIEDI, R., BARANIUK, R., NAVRATIL, J., AND COTTRELL, L. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *PAM* (2003).
- [18] SHRIRAM, A., MURRAY, M., HYUN, Y., BROWNLEE, N., BROIDO, A., FOMENKOV, M., AND KC CLAFFY. Comparison of Public End-to-End Bandwidth Estimation Tools on High-Speed Links. In *PAM 2005*.
- [19] SOMMERS, J., AND BARFORD, P. Self-Configuring Network Traffic Generation. In *IMC* (2004).
- [20] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J., AND BECKER, D. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI* (2002).
- [21] VISHWANATH, K. V., AND VAHDAT, A. Realistic and Responsive Network Traffic Generation. In *ACM SIGCOMM* (2006).
- [22] WILLINGER, W., PAXSON, V., AND TAQQU, M. S. Self-similarity and Heavy Tails: Structural Modeling of Network Traffic. In *A Practical Guide to Heavy Tails: Statistical Techniques and Applications* (1998).

Cutting Corners: Workbench Automation for Server Benchmarking

Piyush Shivam[†] Varun Marupadi⁺ Jeff Chase⁺ Thileepan Subramaniam⁺ Shivnath Babu⁺

[†]*Sun Microsystems*

piyush.shivam@sun.com

⁺*Duke University*

{varun,chase,thilee,shivnath}@cs.duke.edu

Abstract

A common approach to benchmarking a server is to measure its behavior under load from a workload generator. Often a set of such experiments is required—perhaps with different server configurations or workload parameters—to obtain a statistically sound result for a given benchmarking objective.

This paper explores a framework and policies to conduct such benchmarking activities automatically and efficiently. The workbench automation framework is designed to be independent of the underlying benchmark harness, including the server implementation, configuration tools, and workload generator. Rather, we take those mechanisms as given and focus on automation policies within the framework.

As a motivating example we focus on rating the peak load of an NFS file server for a given set of workload parameters, a common and costly activity in the storage server industry. Experimental results show how an automated workbench controller can plan and coordinate the benchmark runs to obtain a result with a target threshold of confidence and accuracy at lower cost than scripted approaches that are commonly practiced. In more complex benchmarking scenarios, the controller can consider various factors including accuracy vs. cost tradeoffs, availability of hardware resources, deadlines, and the results of previous experiments.

1 Introduction

David Patterson famously said:

For better or worse, benchmarks shape a field.

Systems researchers and developers devote a lot of time and resources to running benchmarks. In the lab, they

give insight into the performance impacts and interactions of system design choices and workload characteristics. In the marketplace, benchmarks are used to evaluate competing products and candidate configurations for a target workload.

The accepted approach to benchmarking network server software and hardware is to configure a system and subject it to a stream of request messages under controlled conditions. The workload generator for the server benchmark offers a selected mix of requests over a test interval to obtain an aggregate measure of the server's response time for the selected workload. Server benchmarks can drive the server at varying load levels, e.g., characterized by request arrival rate for open-loop benchmarks [21]. Many load generators exist for various server protocols and applications.

Server benchmarking is a foundational tool for progress in systems research and development. However, server benchmarking can be costly: a large number of runs may be needed, perhaps with different server configurations or workload parameters. Care must be taken to ensure that the final result is statistically sound.

This paper investigates *workbench automation* techniques for server benchmarking. The objective is to devise a framework for an automated *workbench controller* that can implement various policies to coordinate experiments on a shared hardware pool or “workbench”, e.g., a virtualized server cluster with programmatic interfaces to allocate and configure server resources [12, 27]. The controller plans a set of experiments according to some policy, obtains suitable resources at a suitable time for each experiment, configures the test harness (system under test and workload generators) on those resources, launches the experiment, and uses the results and workbench status as input to plan or adjust the next experiments, as depicted in Figure 1. Our goal is to choreograph a set of experiments to obtain a statistically sound result for a high-level objective at low cost, which may involve using different statistical thresholds to balance

This research was conducted while Shivam was a PhD student at Duke University. Subramaniam is currently employed at Riverbed Technologies. This research was funded by grants from IBM and the National Science Foundation through CNS-0720829, 0644106, and 0720829.

cost and accuracy for different runs in the set.

As a motivating example, this paper focuses on the problem of measuring the peak throughput attainable by a given server configuration under a given workload (the *saturation throughput* or *peak rate*). Even this relatively simple objective requires a costly set of experiments that have not been studied in a systematic way. This task is common in industry, e.g., to obtain a qualifying rating for a server product configuration using a standard server benchmark from SPEC, TPC, or some other body as a basis for competitive comparisons of peak throughput ratings in the marketplace. One example of a standard server benchmark is the SPEC SFS benchmark and its predecessors [15], which have been used for many years to establish NFSOPS ratings for network file servers and filer appliances using the NFS protocol.

Systems research often involves more comprehensive benchmarking activities. For example, *response surface mapping* plots system performance over a large space of workloads and/or system configurations. Response surface methodology is a powerful tool to evaluate design and cost tradeoffs, explore the interactions of workloads and system choices, and identify interesting points such as optima, crossover points, break-even points, or the bounds of the effective operating range for particular design choices or configurations [17]. Figure 2 gives an example of response surface mapping using the peak rate. The example is discussed in Section 2. Measuring a peak rate is the “inner loop” for this response surface mapping task and others like it.

This paper illustrates the power of a workbench automation framework by exploring simple policies to optimize the “inner loop” to obtain peak rates in an efficient way. We use benchmarking of Linux-based NFS servers with a configurable workload generator as a running example. The policies balance cost, accuracy, and confidence for the result of each test load, while meeting target levels of confidence and accuracy to ensure statistically rigorous final results. We also show how advanced controllers can implement heuristics for efficient response surface mapping in a multi-dimensional space of workloads and configuration settings.

2 Overview

Figure 1 depicts a framework for automated server benchmarking. An automated *workbench controller* directs benchmarking experiments on a common hardware pool (workbench). The controller incorporates policies that decide which experiments to conduct and in what order, based on the following considerations:

- **Objective.** The controller pursues benchmarking objectives specified by a user. A simple goal might be to obtain a standard NFSOPS rating for a given

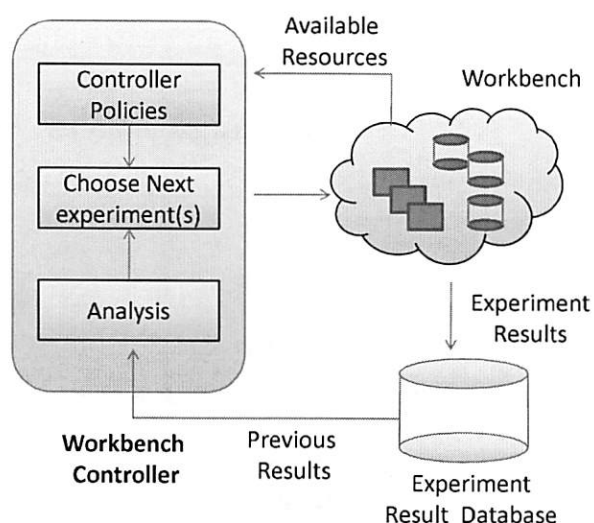


Figure 1: Automated Workbench and Controller.

\vec{W}	read/write ratio, random/sequential ratio, metadata/data ratio, dataset size, file size distribution, directory structure, request mix
\vec{R}	CPU speed, memory size, number of disks
\vec{C}	Number of NFS server I/O daemons (nfsds), type of file system, block size

Table 1: Some workload and configuration factors that affect NFS file server performance.

NFS filer configuration. More complex goals might involve varying the workload or mapping a response surface for different workloads or server configurations. The goals may also specify the response time metric used to obtain the peak rate, and/or thresholds for confidence and accuracy. An objective that we consider is to obtain peak rates with 90% accuracy. An alternative might be to obtain the most complete and/or accurate results achievable within some deadline.

- **Resources.** The controller runs experiments as resources become available. It may tailor the runs to the available resources or schedule multiple runs concurrently.
- **Previous results.** The controller is feedback-driven in that it may consider results of previous runs in designing new experiments. For example, policies in this paper consider the variance of response times at a given test load to determine how many trials are needed to obtain a sound result. The controller can also use results of previous runs to prune the sample space in mapping a response surface.

We characterize the benchmark performance of a server by its *peak rate* or *saturation throughput*, denoted λ^* . λ^* is the highest request arrival rate λ that does not

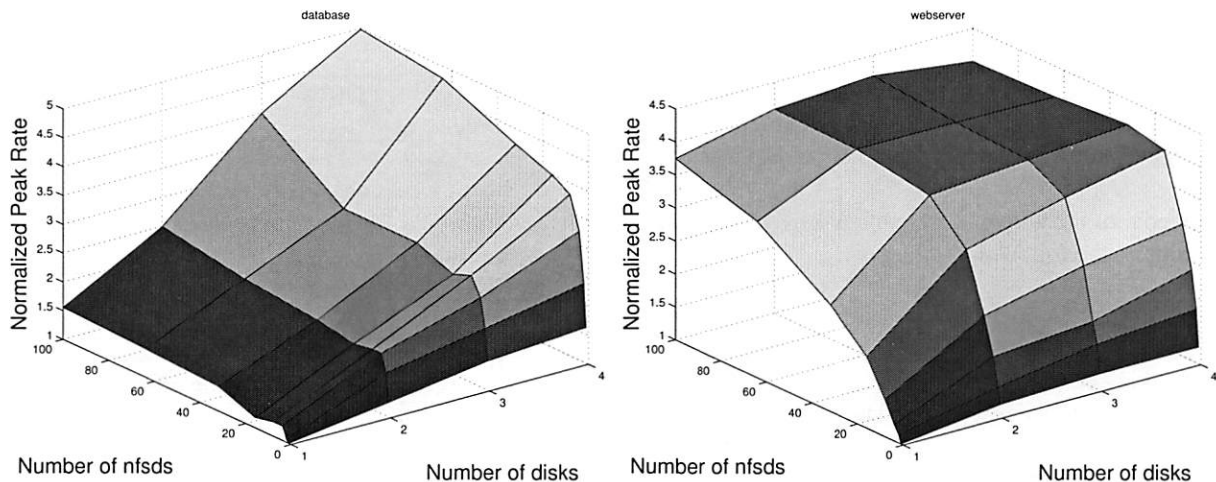


Figure 2: These surfaces illustrate how the peak rate, λ^* , changes with number of disks and number of NFS daemon (nfsd) threads for two canned *fstress* workloads (**DB.TP** and **Web server**) on Linux-based NFS servers. The workloads for this example are described in more detail later, in Table 3.

drive the server into a *saturation state*. The server is said to be in a saturation state if a response time metric exceeds a specified threshold, indicating that the offered load has reached the maximum that the server can process effectively.

The performance of a server is a function of its workload, its configuration, and the hardware resources allocated to it. Each of these may be characterized by a vector of metrics or *factors*, as summarized in Table 1.

Workload \vec{W} . Workload factors define the properties of the request mix and the data sets they operate on, and other workload characteristics.

Configurations (\vec{C}). The controller may vary server configuration parameters (e.g., buffer sizes, queue bounds, concurrency levels) before it instantiates the server for each run.

Resources \vec{R} . The controller can vary the amount of hardware resources assigned to the system under test, depending on the capabilities of the workbench testbed. The prototype can instantiate Xen virtual machines sized along the memory, CPU, and I/O dimensions. The experiments in this paper vary the workload and configuration parameters on a fixed set of Linux server configurations in the workbench.

2.1 Example: NFS Server Benchmarking

This paper uses NFS server benchmarking as a running example. The controllers use a configurable synthetic NFS workload generator called *Fstress* [1], which was developed in previous research. *Fstress* offers knobs for various workload factors (\vec{W}), enabling the controller to configure the properties of the workload's dataset and its request mix to explore a space of NFS workloads. *Fstress* has preconfigured parameter sets that represent standard NFS file server workloads (e.g., SPECsfs97, Postmark),

as well as many other workloads that might be encountered in practice (see Table 3).

Figure 2 shows an example of response surfaces produced by the automated workbench for two canned NFS server workloads representing typical request mixes for a file server that backs a database server (called **DB.TP**) and a static Web server (**Web server**). A response surface gives the response of a metric (peak rate) to changes in the operating range of combinations of factors in a system [17]. In this illustrative example the factors are the number of NFS server daemons (nfsds) and disk spindle counts.

Response surface mapping can yield insights into the performance effects of configuration choices in various settings. For example, Figure 2 confirms the intuition that adding more disks to an NFS server can improve the peak rate only if there is a sufficient number of nfsds to issue requests to those disks. More importantly, it also reveals that the ideal number of nfsds is workload-dependent: standard rules of thumb used in the field are not suitable for all workloads.

2.2 Problem Statement

The challenge for the automated feedback-driven workbench controller is to design a set of experiments to obtain accurate peak rates for a set of test points, and in particular for test points selected to approximate a response surface efficiently.

Response surface mapping is expensive. Algorithm 1 presents the overall benchmarking approach that is used by the workbench controller to map a response surface, and Table 2 summarizes some relevant notation. The overall approach consists of an outer loop that iterates over selected samples from $\langle F_1, \dots, F_n \rangle$, where F_1, \dots, F_n is a subset of factors in the larger $\langle \vec{W}, \vec{R}, \vec{C} \rangle$

space (Step 2). The inner loop (Step 3) finds the peak rate λ^* for each sample by generating a series of test loads for the sample. For each test load λ , the controller must choose the *runlength* r or observation interval, and the *number of independent trials* t to obtain a response time measure under load λ .

The goal of the automated feedback-driven controller is to address the following problems.

1. **Find Peak Rate** (§3). For a given sample from the outer loop of Algorithm 1, minimize the benchmarking cost for finding the peak rate λ^* subject to a target confidence level c and target accuracy a (defined below). Determining the NFSOPS rating of an NFS filer is one instance of this problem.
2. **Map Response Surface** (§4). Minimize the total benchmarking cost to map a response surface for all $\langle F_1, \dots, F_n \rangle$ samples in the outer loop of Algorithm 1.

Minimizing benchmarking cost involves choosing values carefully for the runlength r , the number of trials t , and test loads λ so that the controller converges quickly to the peak rate. Sections 3 and 4 present algorithms that the controller uses to address these problems.

2.3 Confidence and Accuracy

Benchmarking can never produce an exact result because complex systems exhibit inherent variability in their behavior. The best we can do is to make a *probabilistic claim* about the *interval* in which the “true” value for a metric lies based on measurements from multiple independent trials [13]. Such a claim can be characterized by a *confidence level* and the *confidence interval* at this confidence level. For example, by observing the mean response time \bar{R} at a test load λ for 10 independent trials, we may be able to claim that we are 95% confident (the confidence level) that the correct value of \bar{R} for that λ lies within the range $[25ms, 30ms]$ (the confidence interval).

Basic statistics tells us how to compute confidence intervals and levels from a set of trials. For example, if the mean server response time \bar{R} from t trials is μ , and standard deviation is σ , then the confidence interval for μ at confidence level c is given by:

$$\left[\mu - \frac{z_c \sigma}{\sqrt{t}}, \mu + \frac{z_c \sigma}{\sqrt{t}} \right] \quad (1)$$

z_c is a reading from the table of standard normal distribution for confidence level c . If $t \leq 30$, then we use *Student's t* distribution instead after verifying that the t runs come from a normal distribution [13].

The tightness of the confidence interval captures the *accuracy* of the true value of the metric. A tighter bound

λ^*	Peak rate for a given server configuration and workload.
λ	Offered load (arrival rate) for a given test load level.
ρ	Load factor = λ/λ^* for a test load λ .
\bar{R}	Mean server response time for a test load.
R_{sat}	Threshold for \bar{R} at the peak rate: the server is saturated if $\bar{R} > R_{sat}$.
s	Factor that determines the width of the peak-rate region $[R_{sat} \pm sR_{sat}]$ (§3.3).
a	Target <i>accuracy</i> (based on confidence interval width) for the estimated value of λ^* (§2.3).
c	Target <i>confidence level</i> for the estimated λ^* (§2.3).
t	Number of independent trials at a test load.
r	Runlength: the test interval over which to observe the server latency for each trial.

Table 2: Benchmarking parameters used in this paper.

implies that the mean response time from a set of trials is closer to its true value. For a confidence interval $[low, high]$, we compute the percentage accuracy as:

$$accuracy = 1 - error = \left(1 - \frac{high - low}{high + low} \right) \quad (2)$$

3 Finding the Peak Rate

In the inner loop of Algorithm 1, the automated controller searches for the peak rate λ^* for some workload and configuration given by a selected sample of factor values in $\langle F_1, \dots, F_n \rangle$. To find the peak rate it subjects the server to a sequence of test loads $\lambda = [\lambda_1, \dots, \lambda_t]$. The sequence of test loads should converge on an estimate of the peak rate λ^* that meets the target accuracy and confidence.

We emphasize that this step is itself a common benchmarking task to determine a standard rating for a server configuration in industry (e.g., SPECsfs [6]).

3.1 Strawman: Linear Search with Fixed r and t

Common practice for finding the peak rate is to script a sequence of runs for a standard workload at a fixed linear sequence of escalating load levels, with a preconfigured runlength r and number of trials t for each load level. The algorithm is in essence a linear search for the peak rate: it starts at a default load level and increments the load level (e.g., arrival rate) by some fixed increment until it drives the server into saturation. The last load level λ before saturation is taken as the peak rate λ^* . We refer to this algorithm as *strawman*.

Strawman is not efficient. If the increment is too small, then it requires many iterations to reach the peak rate. Its cost is also sensitive to the difference between the peak rate and the initial load level: more powerful server configurations take longer to benchmark. A larger increment

Algorithm 1: Mapping Response Surfaces

- 1) **Inputs:** (a) $\langle F_1, \dots, F_n \rangle$, which is the subset of factors of interest from the full set of factors in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$; (b) Different possible settings of each factor;
 - 2) *// Outer Loop: Map Response Surface.*
foreach distinct sample $\langle F_1 = f_1, \dots, F_n = f_n \rangle$
do
 - 3) *// Inner Loop: Find Peak Rate for the Sample.*
Design a sequence of test loads $[\lambda_1, \dots, \lambda_l]$ to search for the peak rate λ^* ;
foreach test load $\lambda \in [\lambda_1, \dots, \lambda_l]$ **do**
 Choose number of trials t for load λ ;
 Choose runlength r for each trial;
 Configure server and workload generator for the sample; Run t independent trials of length r each, with workload generated at load λ ;
end
 Set $\lambda^* = \lambda$, where $\lambda \in [\lambda_1, \dots, \lambda_l]$ is the largest load that does not take the server to the saturation state;
end
-

can converge on the peak rate faster, but then the test may overshoot the peak rate and compromise accuracy. In addition, *strawman* misses opportunities to reduce cost by taking “rough” readings at low cost early in the search, and to incur only as much cost as necessary to obtain a statistically sound reading once the peak rate is found.

A simple workbench controller with feedback can improve significantly on the *strawman* approach to searching for the peak rate. To illustrate, Figure 3 depicts the search for λ^* for two policies conducting a sequence of experiments, with no concurrent testing. For *strawman* we use runlength $r = 5$ minutes, $t = 10$ trials, and a small increment to produce an accurate result. The figure compares *strawman* to an alternative that converges quickly on the peak rate using binary search, and that adapts r and t dynamically to balance accuracy, confidence, and cost during the search. The figure represents the sequence of actions taken by each policy with cumulative benchmarking time on the x-axis; the y-axis gives the load factor $\rho = \frac{\lambda}{\lambda^*}$ for each test load evaluated by the policies. The figure shows that *strawman* can incur a much higher benchmarking cost (time) to converge to the peak rate and complete the search with a final accurate reading at load factor $\rho = 1$. The *strawman* policy not only evaluates a large number of test loads with load factors that are not close to 1, but also incurs unnecessary

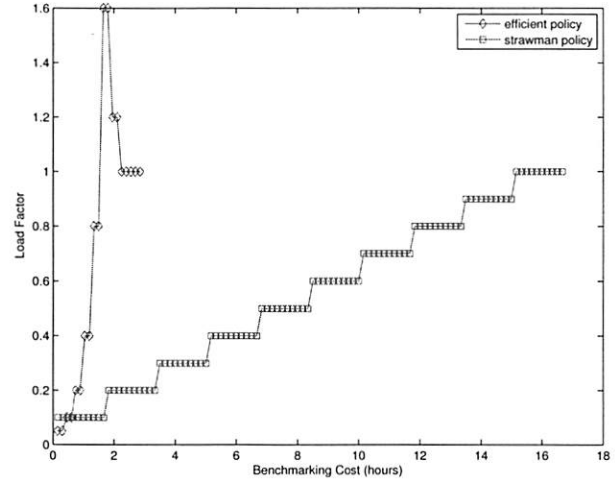


Figure 3: An efficient policy for finding peak rate converges quickly to a load factor near 1, and reduces benchmarking cost by obtaining a high-confidence result only for the load factor of 1. It is significantly less costly than the *strawman* policy: a linear search with a fixed runlength and fixed number of trials per test load.

cost at each load.

The remainder of the paper discusses the improved controller policies in more detail, and their interactions with the outer loop in mapping response surfaces.

3.2 Choosing r and t for Each Test Load

The runlength r and the number of trials t together determine the benchmarking cost incurred at a given test load λ . The controller should choose r and t to obtain the confidence and accuracy desired for each test load at least cost. The goal is to converge quickly to an accurate reading at the peak rate: $\lambda = \lambda^*$ and load factor $\rho = 1$. High confidence and accuracy are needed for the final test load at $\lambda = \lambda^*$, but accuracy is less crucial during the search for the peak rate. Thus the controller has an opportunity to reduce benchmarking cost by adapting the target confidence and accuracy for each test load λ as the search progresses, and choosing r and t for each λ appropriately.

At any given load level the controller can trade off confidence and accuracy for lower cost by decreasing either r or t or both. Also, at a given cost any given set of trials and runlengths can give a high-confidence result with wide confidence intervals (low accuracy), or a narrower confidence interval (higher accuracy) with lower confidence.

However, there is a complication: performance variability tends to increase as the load factor ρ approaches saturation. Figure 4 and Figure 5 illustrate this effect. Figure 4 is a scatter plot of mean server response time (\bar{R}) at different test loads λ for five trials at each load. Note that the variability across multiple trials increases

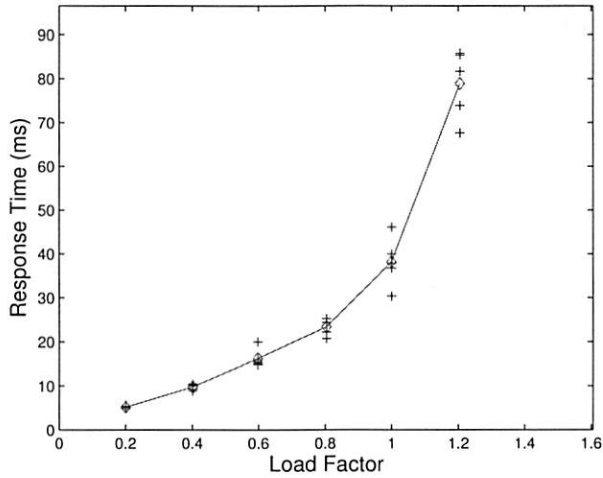


Figure 4: Mean server response time at different test loads for the **DB_TP** *fstress* workload using 1 disk and 4 NFS daemon (nfsd) threads for the server. The variability in mean server response time for multiple trials increases with load. The results are representative of other server configurations and workloads.

as $\lambda \rightarrow \lambda^*$ and $\rho \rightarrow 1$. Figure 5 shows a scatter plot of \bar{R} measures for multiple runlengths at two load factors, $\rho = 0.3$ and $\rho = 0.9$. Longer runlengths show less variability at any load factor, but for a given runlength, the variability is higher at the higher load factor. Thus the cost for any level of confidence and/or accuracy also depends on load level: since variability increases at higher load factors, it requires longer runlengths r and/or a larger number of trials t to reach a target level of confidence and accuracy.

For example, consider the set of trials plotted in Figure 5. At load factor 0.3 and runlength of 90 seconds, the data gives us 70% confidence that $5.6 < \bar{R} < 6$, or 95% confidence that $5 < \bar{R} < 6.5$. From the data we can determine the runlength needed to achieve target confidence and accuracy at this load level and number of trials t : a runlength of 90 seconds achieves an accuracy of 87% with 95% confidence, but it takes a runlength of 300 seconds to achieve 95% accuracy with 95% confidence. Accuracy and confidence decrease with higher load factors. For example, at load factor 0.9 and runlength 90, the data gives us 70% confidence that $21 < \bar{R} < 24$ (93.3% accuracy), or 95% confidence that $20 < \bar{R} < 27$ (85.1% accuracy). As a result, we must increase the runlength and/or the number of trials to maintain target levels of confidence and accuracy as load factors increase. For example, we need a runlength of 120 seconds or more to achieve accuracy $\geq 87\%$ at 95% confidence for this number of trials at load factor 0.9.

Figure 6 quantifies the tradeoff between the runlength and the number of trials required to attain a target accuracy and confidence for different workloads and load factors. It shows the number of trials required to meet

Algorithm 2: Searching for the Peak Rate

- 1) **Initialization.** Peak Rate, $\lambda^* = 0$; Current accuracy of the peak rate, $a_{\lambda^*} = 0$; Current test load, $\lambda_{cur} = 0$; Previous test load, $\lambda_{prev} = 0$;
- 2) Use Algorithm 3 to choose a test load λ by giving current test load λ_{cur} , previous test load λ_{prev} , and mean server response time $\bar{R}_{\lambda_{cur}}$ at λ_{cur} as inputs;
- 3) Set $\lambda_{prev} = \lambda_{cur}$ and $\lambda_{cur} = \lambda$;
- 4) **while** ($a_{\lambda^*} < a$ at confidence c)
- 5) Choose the runlength r for the trial;
- 6) Conduct the trial at λ_{cur} , and measure server response time from this trial, $R_{\lambda_{cur}}$;
- 7) Compute mean server response time at λ_{cur} , $\bar{R}_{\lambda_{cur}}$, from all trials at λ_{cur} . Repeat Step 6 if the number of trials, t , at λ_{cur} is 1;
- 8) Compute confidence interval for the mean server response $\bar{R}_{\lambda_{cur}}$ at target confidence level c ;
- 9) Check for overlap between the confidence interval for $\bar{R}_{\lambda_{cur}}$ and the peak rate region;
- 10) **if** (no overlap with 95% confidence)
 - Go to Step 2 to choose the next test load;
- else**
 - $\lambda^* = \lambda_{cur}$;
 - Compute accuracy a_{λ^*} at confidence c ;
- end**
- end**

an accuracy of 90% at 95% confidence level for different runlengths. The figure shows that to attain a target accuracy and confidence, one needs to conduct more independent trials at shorter runlengths. It also shows a sweet spot for the runlengths that reduces the number of trials needed. A controller can use such curves as a guide to pick a suitable runlength r and number of trials t with low cost.

3.3 Search Algorithm

Our approach uses Algorithm 2 to search for the peak rate for a given setting of factors.

Algorithm 2 takes various parameters to define the conditions for the reported peak rate:

- R_{sat} , a threshold on the mean server response time. The server is considered to be saturated if mean response time exceeds this threshold, i.e., $\bar{R} > R_{sat}$.
- P_{sat} and L_{sat} defining a threshold on percentile server response time. The server is considered to be saturated if the P_{sat} percentile response time exceeds L_{sat} . For example, if $P_{sat} = 0.95$ then the

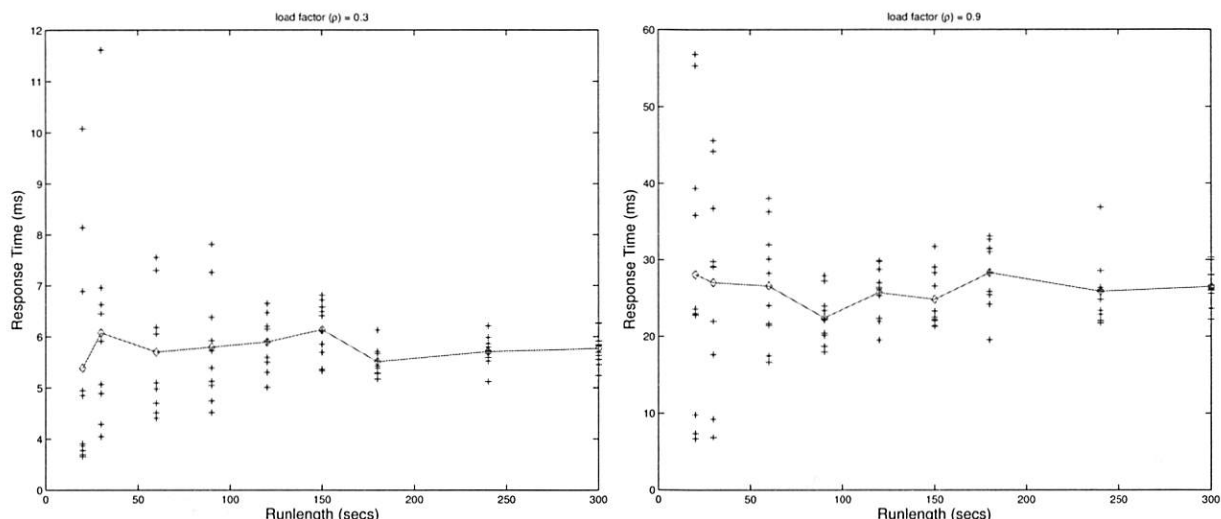


Figure 5: Mean server response time \bar{R} at different workload runlengths for the **DB.TP** *fstress* workload using 1 disk and 4 NFS daemon (nfsd) threads for the server. The variability in mean server response time for multiple trials decreases with increase in runlength. The results are representative of other server configurations and workloads.

server is saturated if no more than 95% of responses show latency at or below L_{sat} . To simplify the presentation we use the R_{sat} threshold test on mean response time and do not discuss P_{sat} further.

- Width parameter s defining the *peak-rate region* $[R_{sat} \pm sR_{sat}]$. The reported peak rate λ^* can be any test load level that drives the mean server response time into this region. (The region $[P_{sat} \pm sP_{sat}]$ is defined similarly.)
- Target confidence c in the peak rate that the algorithm estimates.
- Target accuracy a of the peak rate that the algorithm estimates.

Algorithm 2 chooses (a) a sequence of test loads to try; (b) the number of independent trials at any test load; and (c) the runlength of the workload at that load. It automatically adapts the number of trials at any test load according to the load factor and the desired target confidence and accuracy. At each load level the algorithm conducts a small (often the minimum of two in our experiments) number of trials to establish with 95% confidence that the current test load is not the peak rate (Step 10). However, as soon as the algorithm identifies a test load λ to be a potential peak rate, which happens near a load factor of 1, it spends just enough time to check whether it is in fact the peak rate.

More specifically, for each test load λ_{cur} , Algorithm 2 first conducts two trials to generate an initial confidence interval for $\bar{R}_{\lambda_{cur}}$, the mean server response time at load λ_{cur} , at 95% confidence level. (Steps 6 and 7 in Algorithm 2.) Next, it checks if the confidence interval overlaps with the specified peak-rate region (Step 9).

If the regions overlap, then Algorithm 2 identifies the current test load λ_{cur} as an estimate of a potential peak rate with 95% confidence. It then computes the accuracy of the mean server response time $\bar{R}_{\lambda_{cur}}$ at the current test load, at the target confidence level c (Section 2.1). If it reaches the target accuracy a , then the algorithm terminates (Step 4), otherwise it conducts more trials at the current test load (Step 6) to narrow the confidence interval, and repeats the threshold condition test. Thus the cost of the algorithm varies with the target confidence and accuracy.

If there is no overlap (Step 10), then Algorithm 2 moves on to the next test load. It uses any of several *load-picking* algorithms to generate the sequence of test loads, described in the rest of this section. All load-picking algorithms take as input the set of past test loads and their results. The output becomes the next test load in Algorithm 2. For example, Algorithm 3 gives a load-picking algorithm using a simple binary search.

To simplify the choice of runlength for each experiment at a test load (Step 5), Algorithm 2 uses the “sweet spot” derived from Figure 6 (Section 3.2). The figure shows that for all workloads that this paper considers, a runlength of 3 minutes is the sweet spot for the minimum number of trials.

3.4 The Binsearch Load-Picking Algorithm

Algorithm 3 outlines the *Binsearch* algorithm. Intuitively, Binsearch keeps doubling the current test load until it finds a load that saturates the server. After that, Binsearch applies regular binary search, i.e., it recursively halves the most recent interval of test loads where the algorithm estimates the peak rate to lie.

Binsearch allows the controller to find the lower and

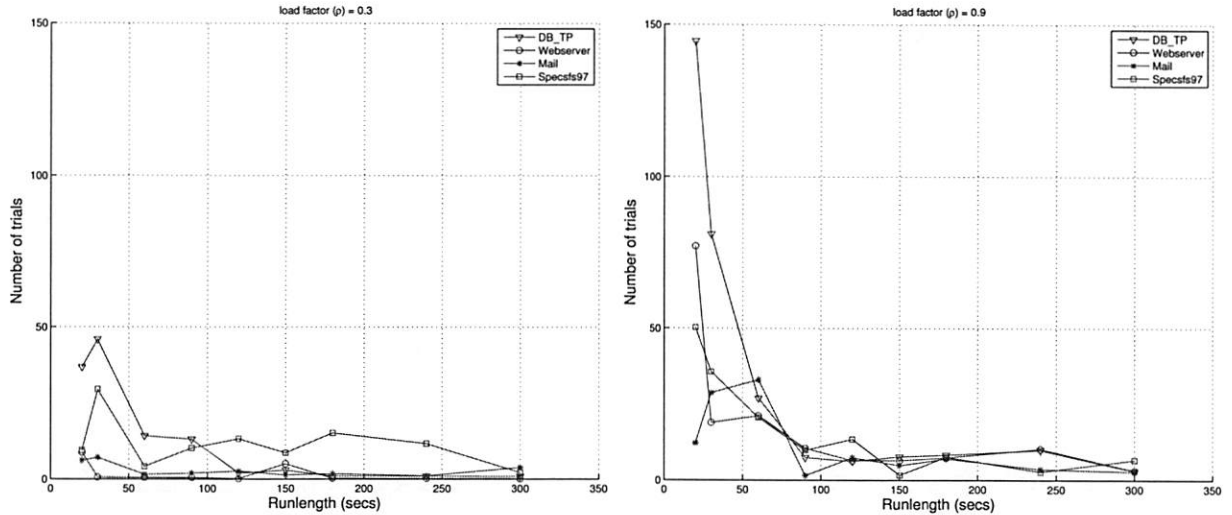


Figure 6: Number of trials to attain 90% accuracy for mean server response time at 95% confidence level at low and high load factors for different runlengths. The results are for server configuration with 1 disk and 4 nfsds, and representative of other server configurations.

Algorithm 3: Binsearch **Input:** Previous load λ_{prev} ; Current load λ_{cur} ; Mean response time $\bar{R}_{\lambda_{cur}}$ at λ_{cur} ; **Output:** Next load λ_{next}

- 1) **Initialization.**
if ($\lambda_{cur} == 0$);
 $\lambda_{next} = 50$ requests/sec;
Phase = Geometric; **Return** λ_{next} ;
 - 2) **Geometric Phase.**
if (Phase == Geometric && $\bar{R}_{\lambda_{cur}} < R_{sat}$)
 $\lambda_{next} = \lambda_{cur} \times 2$;
else
 $\text{binsearch}_{low} = \lambda_{prev}$, and Go to Step 3;
end
 - 3) **Binary Search Phase.**
if ($\bar{R}_{\lambda_{cur}} < R_{sat}$);
 $\text{binsearch}_{low} = \lambda_{cur}$;
else
 $\text{binsearch}_{high} = \lambda_{cur}$;
end
 $\lambda_{next} = (\text{binsearch}_{high} + \text{binsearch}_{low})/2$;
-

upper bounds for the peak rate within a logarithmic number of test loads. The controller can then estimate the peak rate using another logarithmic number of test loads. Hence the total number of test loads is always logarithmic irrespective of the start test load or the peak rate.

3.5 The Linear Load-Picking Algorithm

The *Linear* algorithm is similar to Binsearch except in the initial phase of finding the lower and upper bounds for the peak rate. In the initial phase it picks an increas-

ing sequence of test loads such that each load differs from the previous one by a small fixed increment.

3.6 Model-guided Load-Picking Algorithm

The general *shape* of the response-time vs. load curve is well known, and the form is similar for different workloads and server configurations. This suggests that a model-guided approach could fit the curve from a few test loads and converge more quickly to the peak rate. Using the insight offered by well-known open-loop queuing theory results [13], we experimented with a simple model to fit the curve: $R = 1/(a - b * \lambda)$, where R is the response time, λ is the load, and a and b are constants that depend on the settings of factors in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$. To learn the model, the controller needs tuples of the form $\langle \lambda, R_\lambda \rangle$.

Algorithm 4 outlines the *model-guided* algorithm. If there are insufficient tuples for learning the model, it uses a simple heuristic to pick the test loads for generating the tuples. After that, the algorithm uses the model to predict the peak rate $\lambda = \lambda^*$ for $R = R_{sat}$, returns the prediction as the next test load, and relearns the model using the new $\langle \lambda, R_\lambda \rangle$ tuple at the prediction. The whole process repeats until the search converges to the peak rate. As the controller observes more $\langle \lambda, R_\lambda \rangle$ tuples, the model-fit should improve progressively, and the model should guide the search to an accurate peak rate. In many cases, this happens in a single iteration of model learning (Section 5).

However, unlike the previous approaches, a model-guided search is not guaranteed to converge. Model-guided search is dependent on the accuracy of the model, which in turn depends on the choice of $\langle \lambda, R_\lambda \rangle$ tuples that are used for learning. The choice of tuples is gen-

Algorithm 4: Model-Guided **Input:** Previous loads $\lambda_1, \lambda_2, \dots, \lambda_{cur-1}$; Current load λ_{cur} ; Mean response times $\bar{R}_{\lambda_1}, \bar{R}_{\lambda_2}, \dots, \bar{R}_{\lambda_{cur}}$ at $\lambda_1, \lambda_2, \dots, \lambda_{cur}$; **Output:** Next load λ_{next}

1) **Initialization.**

```

if ( $\lambda_{cur} == 0$ )
    Return  $\lambda_{next} = 50$  requests/sec;
end

if (number of test loads == 1)
    if ( $\bar{R}_{\lambda_{cur}} < R_{sat}$ )
        Return  $\lambda_{next} = \lambda_{cur} \times 2$ ;
    else
        Return  $\lambda_{next} = \lambda_{cur}/2$ ;
    end
end

```

2) **Model Learning and Prediction.**

```

Choose a value of  $\bar{R}_i$  from  $\bar{R}_{\lambda_1}, \dots, \bar{R}_{\lambda_{cur-1}}$  that is
nearest to  $R_{sat}$ . Let the corresponding load be  $\lambda_i$ ;

Learn the model  $R = 1/(a - b\lambda)$  with two tuples
 $\langle \lambda_{cur}, \bar{R}_{\lambda_{cur}} \rangle$  and  $\langle \lambda_i, \bar{R}_i \rangle$ ;

Return  $\lambda_{next} = \frac{R_{sat}a-1}{R_{sat}b}$ ;

```

erated by previous model predictions. This creates the possibility of learning an *incorrect* model which in turn yields incorrect choices for test loads. For example, if most of the test loads chosen for learning the model happen to lie significantly outside the peak rate region, then the model-guided choice of test loads may be incorrect or inefficient. Hence, in the worst case, the search may never converge or converge slowly to the peak rate. We have experimented with other models including polynomial models of the form $R = a + b\lambda + c\lambda^2$, which show similar limitations.

To avoid the worst case, the algorithm uses a simple heuristic to choose the tuples from the list of available tuples. Each time the controller learns the model, it chooses two tuples such that one of them is the last prediction, and the other is the tuple that yields the response time closest to threshold mean server response time R_{sat} . More robust techniques for choosing the tuples is a topic of ongoing study. Section 5 reports our experience with the model-guided choice of test loads. Preliminary results suggest that the model-guided approaches are often superior but can be unstable depending on the initial samples used to learn the model.

3.7 Seeding Heuristics

The load-picking algorithms in Sections 3.5-3.6 generate a new load given one or more previous test loads. How can the controller generate the first load, or *seed*, to try? One way is to use a conservative low load as the seed,

but this approach increases the time spent ramping up to a high peak rate. When the benchmarking goal is to plot a response surface, the controller uses another approach that uses the peak rate of the “nearest” previous sample as the seed.

To illustrate, assume that the factors of interest, $\langle F_1, \dots, F_n \rangle$, in Algorithm 1 are $\langle \text{number of disks, number of nfsds} \rangle$ (as shown in Figure 2). Suppose the controller uses Binsearch with a low seed of 50 to find the peak rate $\lambda_{1,1}^*$ for sample $\langle 1, 1 \rangle$. Now, for finding the peak rate $\lambda_{1,2}^*$ for sample $\langle 1, 2 \rangle$, it can use the peak rate $\lambda_{1,1}^*$ as seed. Thus, the controller can jump quickly to a load value close to $\lambda_{1,2}^*$.

In the common case, the peak rates for “nearby” samples will be close. If they are not, the load-picking algorithms may incur additional cost to recover from a bad seed. The notion of “nearness” is not always well defined. While the distance between samples can be measured if the factors are all quantitative, if there are categorical factors—e.g., file system type—the nearest sample may not be well defined. In such cases the controller may use a default seed or an aggregate of peak rates from previous samples to start the search.

4 Mapping Response Surfaces

We now relate the peak rate algorithm that Section 3 describes to the larger challenge of mapping a peak rate response surface efficiently and effectively, based on Algorithm 1.

A large number of factors can affect performance, so it is important to sample the multi-dimensional space with care as well as to optimize the inner loop. For example, suppose we are mapping the impact of five factors on a file server’s peak rate, and that we sample five values for each factor. If the benchmarking process takes an hour to find the peak rate for each factor combination, then the total time for benchmarking is 130 days. An automated workbench controller can shorten this time by pruning the sample space, planning experiments to run on multiple hardware setups in parallel, and optimizing the inner loop.

We consider two specific challenges for mapping a response surface:

- Algorithm 2 from Section 3.3 is used for the inner loop. However, the algorithm needs a good load-picking policy to generate a sequence of test loads. An efficient controller policy will generate a new test load based on the feedback of the previous results, e.g., the server response time and throughput observed on the earlier test loads. Sections 3.4-3.7 describe the load-picking algorithms we consider.
- Algorithm 1 also depends on a policy to choose the samples in the outer loop. Exhaustive enumeration

of the full factor space in the outer loop can incur an exorbitant benchmarking cost. Depending on the goal of the benchmarking exercise, the controller can choose more efficient techniques.

If the benchmarking objective is to understand the overall trend of how the peak rate is affected by certain factors of interest $\langle F_1, \dots, F_n \rangle$ —rather than finding accurate peak rate values for each sample in $\langle F_1, \dots, F_n \rangle$ —then Algorithm 1 can leverage Response Surface Methodology (RSM) [17] to select the sample points efficiently (in Step 2). RSM is a branch of statistics that provides principled techniques to choose a set of samples to obtain good approximations of the overall response surface at low cost. For example, some RSM techniques assume that a low-degree multivariate polynomial model—e.g., a quadratic equation of the form $\lambda^* = \beta_0 + \sum_{i=1}^n \beta_i F_i + \sum_{i=1}^n \sum_{j=1, j \neq i}^n \beta_{ij} F_i F_j + \sum_{i=1}^n \beta_{ii} F_i^2$ —approximates the surface in the n -dimensional $\langle F_1, \dots, F_n \rangle$ space. This approximation is a basis for selecting a minimal set of samples for the controller to obtain in order to learn a fairly accurate model (i.e., estimate values of the β parameters in the model). We evaluate one such RSM technique in Section 5.

It is important to note that these RSM techniques may reduce the effectiveness of the seeding heuristics described in Section 3.7. RSM techniques try to find sample points on the surface that will add the most information to the model. Intuitively, such samples are the ones that we have the least prior information about, and hence for which seeding from prior results would be least effective. We leave it to future work to explore the interactions of the heuristics for selecting samples efficiently and seeding the peak rate search for each sample.

5 Experimental Evaluation

We evaluate the benchmarking methodology and policies with multiple workloads on the following metrics.

Cost for Finding Peak Rate. Sections 3.3 and 4 present several policies for finding the peak rate. We evaluate those policies as follows:

- The sequence of load factors that the policies consider before converging to the peak rate for a sample. An efficient policy must quickly direct the search to load factors that are near or at 1.
- The number of independent trials for each load factor. The number of trials should be less at low load factors and high around load factor of 1.

Cost for Mapping Response Surfaces. We compare the total benchmarking cost for mapping the response surface across all the samples.

Cost Versus Target Confidence and Accuracy. We demonstrate that the policies adapt the total benchmarking cost to target confidence and accuracy. Higher confidence and accuracy incurs higher benchmarking cost and vice-versa.

Section 5.1 presents the experiment setup. Section 5.2 presents the workloads that we use for evaluation. Section 5.3 evaluates our benchmarking methodology as described above.

5.1 Experimental Setup

Table 1 shows the factors in the $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ vectors for a storage server. We benchmark an NFS server to evaluate our methodology. In our evaluation, the factors in \vec{W} consist of samples that yield four types of workloads: SPECsfs97, Web server, Mail server, and DB_TP (Section 5.2). The controller uses Fstress to generate samples of \vec{W} that correspond to these workloads. We report results for a single factor in \vec{R} : the number of disks attached to the NFS server in $\langle 1, 2, 3, 4 \rangle$, and a single factor in \vec{C} : the number of nfsd daemons for the NFS server chosen from $\langle 1, 2, 4, 8, 16, 32, 64, 100 \rangle$ to give us a total of 32 samples.

The workbench tools can generate both virtual and physical machine configurations automatically. In our evaluation we use physical machines that have 800 MB memory, 2.4 GHz x86 CPU, and run the 2.6.18 Linux kernel. To conduct an experiment, the workbench controller first prepares an experiment by generating a sample in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$. It then consults the benchmarking policy(ies) in Sections 3.4-4 to plot a response surface and/or search for the peak rate for a given sample with target confidence and accuracy.

5.2 Workloads

We use Fstress to generate \vec{W} corresponding to four workloads as summarized in Table 3. A brief summary follows. Further details are in [1].

- **SPECsfs97:** The Standard Performance Evaluation Corporation introduced their System File Server benchmark (SPECsfs) [6] in 1992, derived from the earlier self-scaling LADDIS benchmark [15]. A recent (2001) revision corrected several defects identified in the earlier version [11].
- **Web server:** Several efforts (e.g., [2]) attempt to identify durable characterizations of the Web. We derive the distributions for various parameters and the operation mix from the previous published studies (e.g., [19, 8, 18, 9, 2]).
- **DB_TP:** We model our database workload after TPC-C [7], reading and writing within a few large files in a 2:1 ratio. I/O access patterns are random, with some short (256 KB) sequential asyn-

workload	file popularities	file sizes	dir sizes	I/O accesses
SPECsfs97	random 10%	1 KB – 1 MB	large (thousands)	random r/w
Web server	Zipf ($0.6 < \alpha < 0.9$)	long-tail (avg 10.5 KB)	small (dozens)	sequential reads
DB_TP	few files	large (GB - TB)	small	random r/w
Mail	Zipf ($\alpha = 1.3$)	long-tail (avg 4.7 KB)	large (500+)	seq r, append w

Table 3: Summary of *fstress* workloads used in the experiments.

chronous writes with *commit* (fsync) to mimic batch log writes.

- **Mail:** Electronic mail servers frequently handle many small files, one file per users’ mailbox. Servers append incoming messages, and sequentially read the mailbox file for retrieval. Some users or servers truncate mailboxes after reading. The workload model follows that proposed by Saito et al. [20].

5.3 Results

For evaluating the overall methodology and the policies outlined in Sections 3.3 and 4, we define the peak rate λ^* to be the test load that causes: (a) the mean server response time to be in the [36, 44] ms region; or (b) the 95-percentile request response time to exceed 2000 ms to complete. We derive the [36, 44] region by choosing mean server response time threshold at the peak rate R_{sat} to be 40 ms and the width factor $s = 10\%$ in Table 2. For all results except where we note explicitly, we aim for a λ^* to be accurate within 10% of its true value with 95% confidence.

5.3.1 Cost for Finding Peak Rate

Figure 7 shows the choice of load factors for finding the peak rate for a sample with 4 disks and 32 nfsds using the policies outlined in Section 4. Each point on the curve represents a single trial for some load factor. More points indicate higher number of trials at that load factor. For brevity, we show the results only for **DB_TP**. Other workloads show similar behavior.

For all policies, the controller conducts more trials at load factors near 1 than at other load factors to find the peak rate with the target accuracy and confidence. All policies without seeding start at a low load factor and take longer to reach a load factor of 1 as compared to policies with seeding. All policies with seeding start at a load factor close to 1, since they use the peak rate of a previous sample with 4 disks and 16 nfsds as the seed load.

Linear takes a significantly longer time because it uses a fixed increment by which to increase the test load. However, *Binsearch* jumps to the peak rate region in logarithmic number of steps. The *Model* policy is the quickest to jump near the load factor of 1, but incurs most of its cost there. This happens because the model learned is sufficiently accurate for guiding the search near the

peak rate, but not accurate enough to search the peak rate quickly.

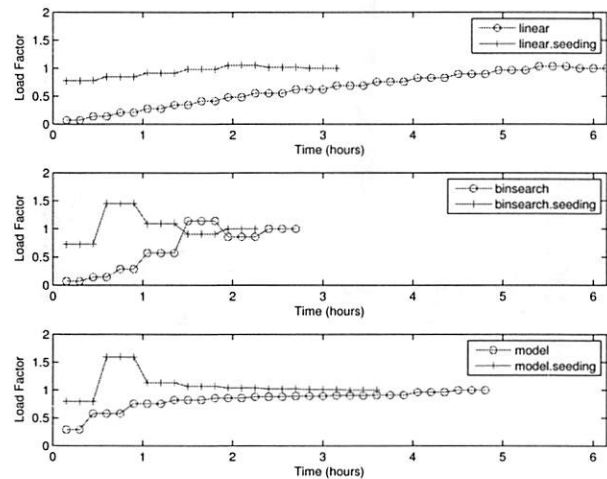


Figure 7: Time spent at each load factor for finding the peak rate for different policies for **DB_TP** with 4 disks and 32 nfsds. Seeded policies were seeded with the peak rate for 4 disks and 16 nfsds. The result is representative of other samples and workloads. All policies except linear quickly converge to the load factor of 1 and conduct more trials there to achieve the target accuracy and confidence.

5.3.2 Cost for Mapping Response Surfaces

Figure 8 compares the total normalized benchmarking cost for mapping the response surfaces for the three workloads using the policies outlined in Section 4. The costs are normalized with respect to the lowest total cost, which is 47 hours and 36 minutes taken by the *Binsearch with Seeding* policy to find the peak rate for **DB_TP**. *Binsearch*, *Binsearch with Seeding*, and *Linear with Seeding* cut the total cost drastically as compared to the linear policy.

We also observe that *Binsearch*, *Binsearch with Seeding*, and *Linear with Seeding* are robust across the workloads, but the model-guided policy is unstable. This is not surprising given that the accuracy of the learned model guides the search. As Section 3.6 explains, if the model is inaccurate the search may converge slowly.

The linear policy is inefficient and highly sensitive to the magnitude of peak rate. The benchmarking cost of *Linear* for **Web server** peaks at a higher absolute value for all samples than for **DB_TP** and **Mail**, causing more than a factor of 5 increase in the total cost for mapping

the surface. Note that for **Mail**, *Binsearch with Seeding* incurs a slightly higher cost than *Binsearch*. For some configurations, as Section 3.7 explains, seeding can incur additional cost to recover from a bad seed resulting in longer search times.

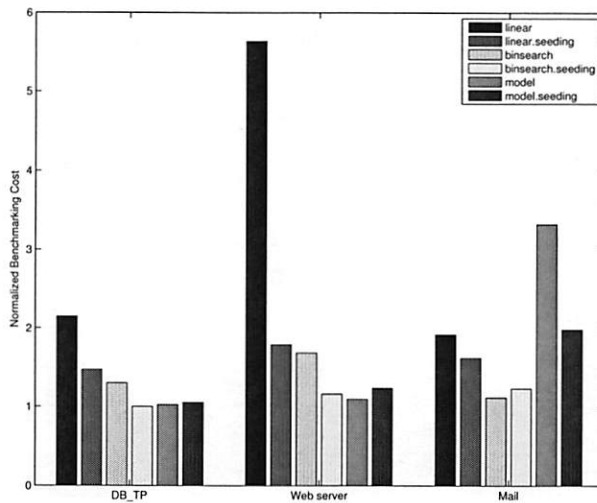


Figure 8: The total cost for mapping response surfaces for three workloads using different policies.

Reducing the Number of Samples. To evaluate the RSM approach presented in Section 4, we approximate the response surface by a quadratic curve in two dimensions: peak rate = func(number of disks, number of nfsds). We use a D-optimal design [17] from RSM to obtain the best of 6, 8 and 10 samples out of a total of 32 samples for learning the response surface equation. We use *Binsearch* to obtain the peak rate for each.

After learning the equation, we use it to predict the peak rate at all the other samples in the surface. Table 4 presents the mean absolute percentage error in predicting the peak rate across all the samples. The results show that D-optimal designs do a very good job of picking appropriate samples, and that very little more can be learned by small increases in the number of points sampled. Improving the accuracy of the surface with limited numbers of sampled points is an area of ongoing research.

Workload	Num. of Samples	MAPE
DB_TP	6, 8, 10	14, 14, 15
Web server	6, 8, 10	9, 9, 9
Mail	6, 8, 10	3.3, 2.8, 2.7

Table 4: Mean Absolute Prediction Error (MAPE) in Predicting the Peak Rate

5.3.3 Cost Versus Target Confidence and Accuracy

Figure 9 shows how the benchmarking methodology adapts the total benchmarking cost to the target confidence and accuracy of the peak rate. The figure shows

the total benchmarking cost for mapping the response surface for the **DB_TP** using the *Binsearch* policy for different target confidence and accuracy values.

Higher target confidence and accuracy incurs higher benchmarking cost. At 90% accuracy, note the cost difference between the different confidence levels. Other workloads and policies exhibit similar behavior, with **Mail** incurring a normalized benchmarking cost of 2 at target accuracy of 90% and target confidence of 95%.

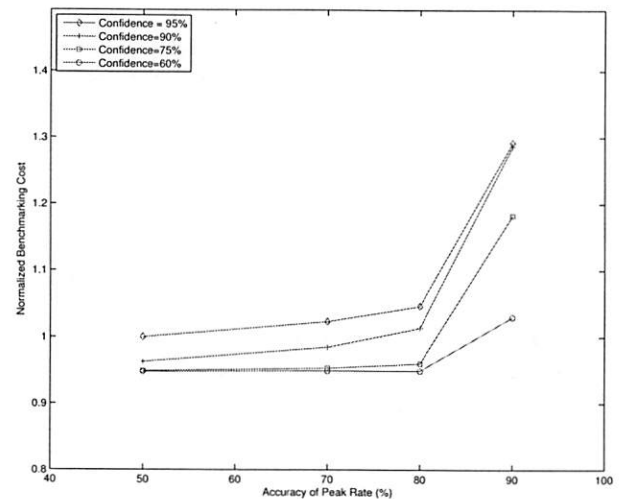


Figure 9: The total benchmarking cost adapts to the desired confidence and accuracy. The cost is shown for mapping the response surface for **DB_TP** using the *Binsearch* policy. Other workloads and policies show similar results.

So far, we configure the target accuracy of the peak rate by configuring the accuracy, a , of the response time at the peak rate. The width parameter s also controls the accuracy of the peak rate (Table 2) by defining the peak rate region. For example, $s = 10\%$ implies that if the mean server response time at a test load is within 10% of the threshold mean server response time, R_{sat} , then the controller has found the peak rate. As the region narrows, the target accuracy of the peak rate region increases. In our experiments so far, we fix $s = 10\%$.

Figure 10 shows the benchmarking cost adapting to the target accuracy of the peak rate region for different policies at a fixed target confidence interval for **DB_TP** ($c = 95$) and fixed target accuracy of the mean server response time at the peak rate ($a = 90\%$). The results for other workloads are similar. All policies except the model-guided policy incur the same benchmarking cost near or at the peak rate since all of them do binary search around that region. Since a narrower peak rate region causes more trials at or near load factor of 1, the cost for these policies converge.

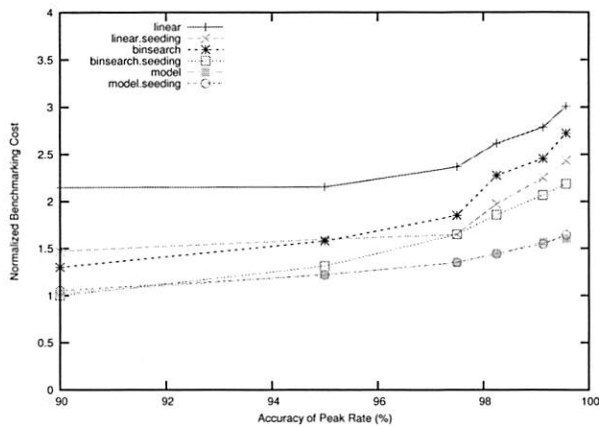


Figure 10: Benchmarking cost adapts to the target accuracy of the peak rate region for all policies. As the region narrows, the majority of the cost is incurred at or near the peak rate. Linear and Binsearch incur the same cost close to the peak rate, and hence their cost converges as they conduct more trials near the peak rate. The cost is shown for **DB_TP**. Other workloads show similar results.

6 Related Work

Several researchers have made a case for statistically significant results from system benchmarking, e.g., [4]. Auto-pilot [26] is a system for automating the benchmarking process: it supports various benchmark-related tasks and can modulate individual experiments to obtain a target confidence and accuracy. Our goal is to take the next step and focus on an automation framework and policies to orchestrate sets of experiments for a higher level benchmarking objective, such as evaluating a response surface or obtaining saturation throughputs under various conditions. We take the workbench test harness itself as given, and our approach is compatible with advanced test harnesses such as Auto-pilot.

While there are large numbers and types of benchmarks, (e.g., [5, 14, 3, 15]) that test the performance of servers in a variety of ways, there is a lack of a general benchmarking methodology that provides benchmarking results from these benchmarks efficiently with confidence and accuracy. Our methodology and techniques for balancing the benchmarking cost and accuracy are applicable to all these benchmarks.

Zadok et al. [25] present an exhaustive nine-year study of file system and storage benchmarking that includes benchmark comparisons, their pros and cons [22], and makes recommendations for systematic benchmarking methodology that considers a range of workloads for benchmarking the server. Smith et al. [23] make a case for benchmarks that capture composable elements of realistic application behavior. Ellard et al. [10] show that benchmarking an NFS server is challenging because of the interactions between the server software configurations, workloads, and the resources allocated to the

server. One of the challenges in understanding the interactions is the large space of factors that govern such interactions. Our benchmarking methodology benchmarks a server across the multi-dimensional space of workload, resource, and configuration factors efficiently and accurately, and avoids brittle “claims” [16] and “lies” [24] about a server performance.

Synthetic workloads emulate characteristics observed in real environments. They are often self-scaling [5], augmenting their capacity requirements with increasing load levels. The synthetic nature of these workloads enables them to preserve workload features as the file set size grows. In particular, the SPECsfs97 benchmark [6] (and its predecessor LADDIS [15]) creates a set of files and applies a pre-defined mix of NFS operations. The experiments in this paper use Fstress [1], a synthetic, flexible, self-scaling NFS workload generator that can emulate a range of NFS workloads, including SPECsfs97. Like SPECsfs97, Fstress uses probabilistic distributions to govern workload mix and access characteristics. Fstress adds file popularities, directory tree size and shape, and other controls. Fstress includes several important workload configurations, such as Web server file accesses, to simplify file system performance evaluation under different workloads [23] while at the same time allowing standardized comparisons across studies.

Server benchmarking isolates the performance effects of choices in server design and configuration, since it subjects the server to a steady offered load independent of its response time. Relative to other methodologies such as application benchmarking, it reliably stresses the system under test to its saturation point where interesting performance behaviors may appear. In the storage arena, NFS server benchmarking is a powerful tool for investigation at all layers of the storage stack. A workload mix can be selected to stress any part of the system, e.g., the buffering/caching system, file system, or disk system. By varying the components alone or in combination, it is possible to focus on a particular component in the storage stack, or to explore the interaction of choices across the components.

7 Conclusion

This paper focuses on the problem of workbench automation for server benchmarking. We propose an automated benchmarking system that plans, configures, and executes benchmarking experiments on a common hardware pool. The activity is coordinated by an automated controller that can consider various factors in planning, sequencing, and conducting experiments. These factors include accuracy vs. cost tradeoffs, availability of hardware resources, deadlines, and the results reaped from previous experiments.

We present efficient and effective controller policies

that plot the saturation throughput or peak rate over a space of workloads and system configurations. The overall approach consists of iterating over the space of workloads and configurations to find the peak rate for samples in the space. The policies find the peak rate efficiently while meeting target levels of confidence and accuracy to ensure statistically rigorous benchmarking results. The controller may use a variety of heuristics and methodologies to prune the sample space to map a complete response service, and this is a topic of ongoing study.

References

- [1] D. C. Anderson and J. S. Chase. Fstress: A flexible network file service benchmark. Technical Report CS-2002-01, Duke University, Department of Computer Science, January 2002.
- [2] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, April 1996.
- [3] T. Bray. Bonnie file system benchmark, 1996. <http://www.textuality.com/bonnie>.
- [4] A. B. Brown, A. Chanda, R. Farrow, A. Fedorova, P. Maniatis, and M. L. Scott. The many faces of systems research: And how to evaluate them. In *Proceedings of the 10th conference on Hot Topics in Operating Systems*, June 2005.
- [5] P. Chen and D. Patterson. A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performance. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.
- [6] S. P. E. Corporation. SPEC SFS release 3.0 run and report rules, 2001.
- [7] T. P. P. Council. TPC benchmark C standard specification, August 1992. Edited by François Raab.
- [8] M. Crovella, M. Taqqu, and A. Bestavros. In *A Practical Guide To Heavy Tails*, chapter 1 (Heavy-Tailed Probability Distributions in the World Wide Web). Chapman & Hall, 1998.
- [9] R. Doyle, J. Chase, S. Gadde, and A. Vahdat. The trickle-down effect: Web caching and server request distribution. In *Proceedings of the Sixth International Workshop on Web Caching and Content Delivery*, June 2001.
- [10] D. Ellard and M. Seltzer. NFS tricks and benchmarking traps. In *Proceedings of the FREENIX 2003 Technical Conference*, June 2003.
- [11] S. Gold. Defects in SFS 2.0 which affect the working-set, July 2001. http://www.spec.org/osg/sfs97/sfs97_defects.html.
- [12] D. Irwin, J. S. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proc. of the USENIX Annual Technical Conf.*, Jun 2006.
- [13] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, May 1991.
- [14] J. Katcher. Postmark: A new file system benchmark. Technical Report 3022, Network Appliance, October 1997.
- [15] B. Keith and M. Wittle. LADDIS: The next generation in NFS file server benchmarking. In *Proceedings of the USENIX Annual Technical Conference*, June 1993.
- [16] J. C. Mogul. Brittle metrics in operating systems research. In *Proceedings of the the 7th Workshop on Hot Topics in Operating Systems*, March 1999.
- [17] R. H. Myers and D. C. Montgomery. *Response Surface Methodology: Process and Product in Optimization Using Designed Experiments*. John Wiley & Sons, Inc., New York, NY, USA, 1995.
- [18] National laboratory for applied network research (NLNRR). <http://moat.nlanr.net>.
- [19] C. Roadknight, I. Marshall, and D. Vearer. File popularity characterisation. In *Proceedings of the 2nd Workshop on Internet Server Performance*, May 1999.
- [20] Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, December 1999.
- [21] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, April 2006.
- [22] C. Small, N. Ghosh, H. Saleed, M. Seltzer, and K. Smith. Does systems research measure up. Technical Report TR-16-97, Harvard University, Department of Computer Science, November 1997.
- [23] K. A. Smith. *Workload-Specific File System Benchmarks*. PhD thesis, Harvard University, Cambridge, MA, January 2001.
- [24] D. Tang and M. Seltzer. Lies, Damned Lies, and File System Benchmarks. In *VINO: The 1994 Fall Harvest*. Harvard Division of Applied Sciences Technical Report TR-34-94, December 1994.
- [25] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A nine year study of file system and storage benchmarking. Technical Report FSL-07-01, Computer Science Department, Stony Brook University, May 2007.
- [26] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, April 2005.
- [27] A. Yumerefendi, P. Shivam, D. Irwin, P. Gunda, L. Grit, A. Demberel, J. Chase, and S. Babu. Towards an autonomic computing testbed. In *Proceedings of the Workshop on Hot Topics in Autonomic Computing*, June 2007.

Power-aware Remote Replication for Enterprise-level Disaster Recovery Systems

Kazuo Goda

Masaru Kitsuregawa

Institute of Industrial Science, the University of Tokyo, Japan

{kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract

Electric energy consumed in data centers is rapidly growing. Power-aware IT, recently called ‘green IT’, is widely recognized as a significant challenge. Disk storage is a non-negligible energy consumer. Rather, in light of recent data-intensive systems where a number of disk drives are incorporated, the disk storage may be what we must consider primarily. Yet, all of the disk drives are not used for primary datasets, but rather larger portion of them are utilized for storing a variety of *copies* such as backups and snapshots. Saving the energy of such storage resources that manage copies is a promising approach. The paper presents a power-aware disaster recovery system, in which the reflection of transferred updated information can be deferred through eager compaction technique. Great energy saving of storage systems is expected in the remote secondary site. Our experiments using a commercial database system show that 80-85% energy of the secondary-site disk storage can be saved with small penalties of possible service breakdown time.

1 Introduction

Many attentions are paid on the energy consumption of IT systems, which has been grown up by 25% every year [9]. The recent analysis [2] reports that the annual electricity cost paid by the system owner will go up to twice higher than the annual server expense in 2009. More and more powerful cooling systems and power-supply equipments are being installed into data centers for accommodating the increase of energy consumption; accordingly, the electric energy and the related equipments account for 44% of TCO in a typical system [1]. In addition to the cost issue, energy and heat management has become a key of data center design and operation. The exploding energy consumption might strictly constrain the design space of modern IT systems [32]. Hence, energy saving is a grand challenge for IT research and development.

Storage systems are non-negligible energy consumer in IT systems. Storage systems present 27% of total energy consumption in a typical data center [23]. As the digital data volume is explosively increasing [16], extremely many disk drives are being incorporated into an enterprise system to improve the throughput. Thus, much larger portion of the total energy may be consumed by the storage system in high-performance data-intensive systems. Q. Zhu et al. in paper [34] points out an interesting example, where disk drives account for 71% of the total energy consumption in a large-scale OLTP system. Therefore, energy saving of the disk storage is rather essential as well as server processors and network devices.

Interestingly, all the disk drives of recent enterprise disk storage are not necessarily used for primary datasets. Rather, larger portion of the disk drives are utilized for storing a variety of *copies* to improve the system performance and availability. Suppose a simple IT system, which holds a single snapshot in a local data center and a backup copy in a remote data center. Two thirds of all the disk drives equipped in the total system are used for copies. Modern enterprise systems may use much more disk drives for copies [18]. Saving the energy of such storage resources should be a natural idea.

The paper proposes a power-aware disaster recovery system. It has been widely recognized that the business breakdown due to unpredictable disasters such as terrors and hurricanes provides a nation and a society with terrible damage [7, 25]. Business continuity is being enforced by nation-level legal systems as well as by enterprise-level internal disciplines [3, 26, 29]. The disaster recovery system [8, 15, 17] is a practical solution which places a remote secondary site and, in case of disaster, continues the business on the secondary site. By concentrating transferred update information through eager compaction technique, our proposed system can derive longer idle time of the data volume in order to reduce significantly the energy consumption of disk storage of the secondary site with small penalties of service break-

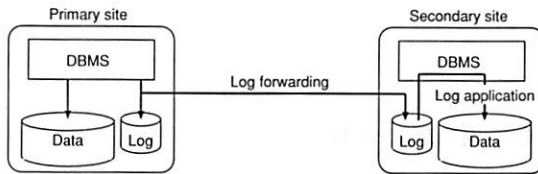


Figure 1: A disaster recovery system based on database log forwarding.

down time. In the real disaster recovery system, many resources of the secondary site may not be fully utilized when the primary site is normally operating. Great energy saving is expected in practice. To the best of our knowledge, similar researches have not been published.

This paper is organized as follows. Section 2 concisely describes disaster recovery systems that are currently deployed in many enterprise systems. Section 3 proposes novel techniques for energy saving of disaster recovery systems and Section 4 evaluates the proposal through the experiments using a commercial database system. Section 5 briefly summarizes related works and finally Section 6 concludes the paper.

2 Disaster recovery system

A disaster recovery system comprises two or more sites. Business is usually operated in the primary site and, once a disaster damages the primary site, the business is continued in the remote secondary site. For enabling such disaster recovery, up-to-date data of the primary site must be always copied into the secondary site. Many solutions have been proposed in papers and deployed into real systems, but the basic idea is similar in that they are composed of the following two steps: (1) transferring only updated information of the primary site to the secondary site and (2) reflecting it to the storage in the secondary site. Here the updated information means queries or transactions for the conventional logical database replication, changed blocks for the storage-level physical block forwarding [8, 17], and database log entries for the log forwarding [15, 27].

The recovery capability of such a disaster recovery system can be defined by two metrics: recovery point objective (RPO) and recovery time objective (RTO). RPO denotes possibility of data loss, i.e. how latest data can be recovered in case of disaster. RTO means inter-site takeover overhead, i.e. how soon the business can start again in the secondary site in case of disaster. It is preferable that RPO and RTO would be small. This paper focuses on enterprise-level systems such as brokerage and e-commerce, which accept only small service breakdown time and do never allow any data loss even in case of disaster. Thus, we assume here that inter-site data transfer

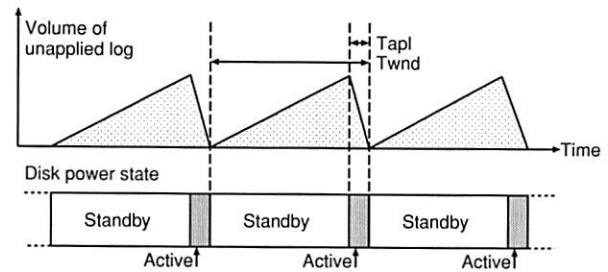


Figure 2: Deferred log application.

is done in the synchronous fashion; specifically, RPO is always zero. Needless to say, our contribution can be directly applied to more relaxed asynchronous modes.

3 Power-aware disaster recovery system

To save the energy consumption of disaster recovery systems, we focus on the disk storage in the secondary site, in which most of the storage resources are used only for storing backup copies.

Let us describe a scenario based on database log forwarding, which is recently deployed in high-end disaster recovery systems. Figure 1 illustrates a disaster recovery system based on database log forwarding, where physical database log is shipped from the primary site to the secondary site and the forwarded log is applied in the secondary site.

3.1 Deferred log application

Our idea is to *defer* the application of transferred database log to derive longer idle time of the data volume in the secondary site. Figure 2 illustrates a batch application scheme for realizing such deferred log application. In the secondary site, the transferred log is stored in the log volume and is not immediately applied to the data volume. While the database log is not being applied, the data volume is idle, so that the energy consumption of the data volume can be saved by spinning down the volume. Longer standby time gives greater energy saving, but provides larger amount of unapplied log stored in the log volume. In case of disaster, the secondary site must apply all the unapplied log before starting the business again. Therefore, deferability of log application is mainly determined by RTO requirements.

Let us discuss the relationship between RTO requirements and deferability. Let R_{gen} and R_{apl} be the log generation rate in the primary site and the maximum log application rate in the secondary site respectively. We assume that the recovery time of the secondary site is proportional to the amount of unapplied log¹. The fol-

¹Fast log application is a key to quick recovery [19]. Other marginal

Table 1: Typical OLTP systems

Rank	Vendor	System	tpmC	Database	# of disks (data volume)	# of disks (log volume)
1	IBM	System p5 595	4,033,378	IBM DB2 9	6400	360
2	IBM	eServer p5 595	3,210,540	IBM DB2 UDB 8.2	6400	140
3	IBM	eServer p5 595	1,601,784	Oracle Database 10g	3200	96
4	Fujitsu	PRIMEQUEST 540 16p/32c	1,238,579	Oracle Database 10g	1920	224
5	HP	Integrity Superdome 64p/64c	1,231,433	Microsoft SQL Server 2005	1680	56

Quoted from *Top Ten TPC-C by Performance Version 5 Results* disclosed at <http://www.tpc.org/> as of December 11, 2006.

lowing formulae give the optimal batch configuration: a batch interval T_{wnd} and a necessary log application time in the interval T_{apl} , the combination of which can gain maximum energy conservation. For the page limitation, we have to omit the mathematical proof.

$$T_{wnd} = \frac{R_{apl}^2}{(R_{apl} - R_{gen}) \cdot R_{gen}} T_{RTO}$$

$$T_{apl} = \frac{R_{apl}}{R_{apl} - R_{gen}} T_{RTO}$$

Here T_{RTO} denotes a RTO requirement, i.e. given allowance of service breakdown time. Implicitly, $R_{apl} > R_{gen}$ and $T_{RTO} > T_{up} + T_{down}$, where T_{up} and T_{down} denote time penalties of spinning up and down respectively. The secondary site can concentrate log application based on the above batch configuration and proactively spin up and down the data volume in order to save energy consumption of the disk storage.

3.2 Eager log compaction

Obviously, larger $\frac{R_{apl}}{R_{gen}}$ leads to greater energy saving. In this section, we introduce *eager compaction* technique to improve the log application throughput significantly.

In a disaster recovery system based on log forwarding, the transferred log entries are applied to the data volume in a way similar to database redo operation. In the normal redo operations, log entries are applied strictly in log sequence number (LSN) order. On the contrary, our proposed eager techniques can compact the log sequence in a window buffer and apply the compacted sequence to the data volume. The compaction process comprises *log folding* and *log sorting*. Log folding is a technique to reduce the number of log entries to be applied. The log entries which manipulate the identical record are coalesced in the window buffer. For example, assuming that three log entries, `insert(data1)`, `update(data1 → data2)` and `update(data2 → data3)`, are given in the sequence, these three entries, manipulating the same record, can be logically folded into a single entry, `insert(data3)`. On the other hand, log sorting reorders log entries in the window buffer to im-

recovery overheads are out of the scope of this paper.

prove disk access sequentiality. In many cases, an entry of database log has a physical reference to the target record. Log sorting leverages such physical information. Both the methods together can improve the log application throughput. Accordingly, larger energy saving can be expected.

3.3 Discussion

Here we would like to briefly discuss our contribution to the total energy consumed by the secondary-site storage. Table 1 shows top-five systems quoted from “Top Ten TPC-C by Performance Version 5 Results”. Four systems used only 2.2-5.6% of all the disk drives for database log, and the other system used only 11.6% for log. That is, in the secondary site, only very few disk drives must be always spinning actively and the other disk drives can be spun down by the combination of deferred log application and eager log compaction. The contribution of our idea is still significant on the whole storage system in the secondary site.

Although this section discusses the problem mainly based on the database log forwarding, the proposed method can be easily extended to other remote replication methods such as logical database replication and physical block forwarding. Specifically, eager compaction technique can be directly applied to physical block forwarding. Forwarded blocks can be folded and sorted similarly based on physical block address. On the other hand, slight modification is necessary for logical database replication, since queries and transactions described in SQLs are not aware of physical addresses. Queries and transactions should be scheduled with assistance of batch query scheduling techniques [21, 24]. So far, such compaction techniques of updated information were intended for reducing inter-site traffic [17]. In contrast, our attempt is focused on deriving long idle period for energy saving.

4 Evaluation

This section presents validating experiments using TPC-C benchmark, showing that disk power consumption of the secondary site can be saved with slight degradation of the quality of business continuity. Online transactions

Table 2: Basic parameters of disk drive models.

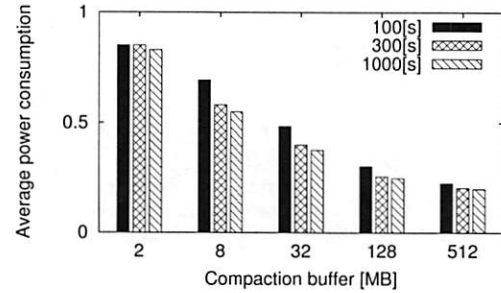
Model	IBM Ultrastar 36Z15	HGST Deskstar T7K250
Capacity	18.4 GB	250 GB
Rotational speed	15000 rpm	7200 rpm
Avg. seek time	3.4 ms	8.5 ms
Transfer rate	55 MB/s	61 MB/s
Active power	39.0 W	9.7 W
Idle power	22.3 W	5.24 W
Standby power	4.15 W	(U) 4.04 W (L) 2.72 W (N) 0.93 W
Spin-down penalties (time and energy)	15.0 s 62.25 J	(U) 0.7 s, 3.5 J (L) 17.0 s, 19.0 J (N) 0.7 s, 3.5 J
Spin-up penalties (time and energy)	26.0 s 904.8 J	(U) 0.7 s, 3.5 J (L) 17.0 s, 19.0 J (N) 0.7 s, 3.5 J

(U): unloaded mode, (L): low-rpm mode, (N): non-spinning mode

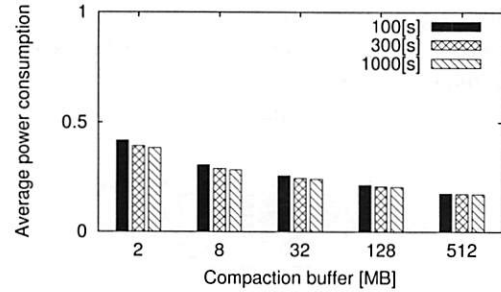
are typical workloads that are seen in enterprise-level disaster recovery systems.

We prepared a hybrid simulation environment for measuring the potential energy saving due to the proposed system. In the experiment, we used a disk drive simulator which can calculate energy consumption based on a disk drive model. We implemented deferred log application and eager log compaction on the top of the disk drive simulator. The developed log applier can apply database log generated by HiRDB [15], a commercial database system.

The experiments were done on a Linux server with dual Xeon processors and 2GB main memory. We set up TPC-C benchmark with 16 and 160 warehouses respectively, and we generated database log on each configuration by executing one million transactions using HiRDB plus 512MB database buffer with no think time. At this execution, we also traced IO behavior by using a kernel-level IO tracer. Then, by replaying the traced IOs using a disk drive model in the simulation environment, we simulated log generation at the primary site. Here, we assumed that the primary site processed transactions at the maximum rate on the specified disk drive model. Next, we applied the generated database log by the log applier, and measured the power reduction effect at the secondary site. Throughout this experiment, we followed the storage system configuration of "IBM System p5 595" in Table 1. That is, we assumed that 94.4% of disk drives were used for the data volume and the same type of disk storage was used both in the primary and secondary sites. The experiments were conducted for different RTO requirements and different window buffer lengths. For validation, we compared the energy saving of the proposed power-aware system with the conventional system in which the transferred update information is immediately reflected.



(a) 16 warehouses



(b) 160 warehouses

Figure 3: Power saving of secondary-site disk storage with high-end disk drives.

Table 3: Batch interval for 160 warehouses and high-end disks under 100 seconds of RTO requirement.

Window buffer	32 MB	128 MB	512 MB
Batch interval	536 s	1070 s	2150 s

Figure 3 summarizes the results obtained with a high-end disk drive model. Basic parameters of the model are presented in Table 2. This model, which is based on IBM Ultrastar 36Z15, may not be new, but has been used in many previous papers [4, 20, 33, 34]. In the graphs, each bar, denoting the average power consumption of the disk storage in the secondary-site storage, is normalized by that of the conventional system. Larger window buffer could accelerate the log application throughput more, and accordingly, greater power saving was gained. Note that, by using 512 MB window buffer, which is as large as the database buffer of the primary site, 85% of the power could be conserved totally in the secondary-site storage. Such great saving was supported by accelerated log application; $\frac{R_{apl}}{R_{gen}}$ could speed up to 20.5 (W=16) and 49.3 (W=160) at maximum by eager log compaction. On the other hand, more tolerant RTO requirements could lead to more energy saving, but its contribution was slight. In our experiments, only short RTOs (such as 30 seconds) failed because RTOs were shorter than time penalties of spinning up and down the volume, but moderate RTOs (100 seconds and more) could conserve the energy so much.

Let us consider the batch interval T_{wnd} . Frequent transition of energy modes affects the life time of disk drives. Table 3 summaries sampled values of T_{wnd} . With small buffer, the data volume had to change its modes frequently, but with as large buffer as 512 MB, the disk mode changes only 40 times a day. Note that this frequency was given when the primary site generates database log at top speed. Thus, it looks almost acceptable, since many high-end disk drives support at least 50,000 cycles of starts/stops. Of course, more tolerant RTO gives larger intervals. This analysis reveals that eager compaction is a key technique to the longevity of disk drives.

We conducted the experiments using a recent mid-range disk drive, which has new energy-efficient features [14]. Basic parameters of the model are presented in Table 2 too. The disk drive, which is based on HGST Deskstar T7K250, has three standby states: unload, low-rpm and non-spinning (equal to conventional standby mode). Basically, the proposed disaster recovery system could work with mid-range disk drives similarly. But, since recent mid-range disk drives can change the energy modes with much smaller time penalties than high-end disk drives, the proposed system could work for such small RTOs as 30 seconds. Figure 4 compares these three standby modes with 160 warehouses. By using the non-spinning standby mode, 80% saving was gained at maximum in comparison with the conventional system. However, we cannot observe the substantial benefit of using new energy-saving functions such as unload and low-rpm.

In summary, the proposed power-aware disaster recovery system can achieve great energy saving of the secondary-site disk storage without little harm to the recovery capability. Only 100 seconds and 30 seconds of RTO allowance were needed for high-end disks and mid-range disks respectively. This observation is surprising, since strict high-availability systems, as known as five nines (99.999%), allow only 315 seconds of breakdown per year. Our proposal can be promising in such top-drawn disaster recovery systems.

5 Related works

Research communities have presented various attempts for energy-efficient disk storage.

The simplest approach is to transition disk drives to a low-power mode after the predetermined time period has elapsed after the last disk access. This technique is widely deployed in commercial disk drives. More sophisticated techniques that try to tune the threshold adaptively have been also studied [6, 11]. Such threshold based techniques work effectively for battery-operated mobile and laptop computers. However, it looks difficult

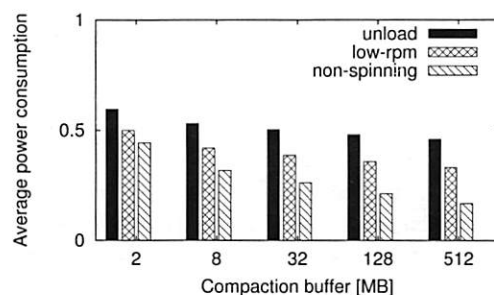


Figure 4: Power saving of secondary-site disk storage with mid-range disk drives under 30 seconds of RTO requirement.

to directly apply these techniques to enterprise systems.

Massive Array of Idle Disks (MAID) [5] and Popular Data Concentration (PDC) [4] are alternative approaches that migrate/replicate popular blocks on specific disk drives to create long idle period of the other disk drives. These techniques leveraging access locality are deployed in real archival storage systems.

Exploiting redundancy information and large cache space that RAID capability holds seems a reasonable approach. Energy Efficient RAID (EERAID) [20] and RIMAC [33] can arrange IO requests at RAID controllers so as to avoid evicting out blocks that are originally stored in spun-down disk drives as much as possible. Power-Aware RAID (PARAID) [30] introduces an asymmetric parity placement on the legacy RAID-5 so that the system can dynamically change the number of actively spinning disk drives.

Other researchers [12, 34] have actively studied on multi-speed disk drives which have the capability of changing the rotational speeds. These attempts look very effective. However, to our knowledge, such multi-speed disk drives are still limited in experimental prototypes and not yet seen in the market.

Recently several application-assisted approaches for storage energy conservation have been reported. Cooperative IO [22, 31] is a set of power-aware IO system calls, by which the user can specify deferability and abortability to each IO. Compiler-based application transformation [10, 13, 28] tries to arrange IO commands in source code levels in order to concentrate IO requests.

Our work differs from these previous works in that we are trying to fully leverage the characteristics of the secondary-site disk storage. That is, the disk storage there manages only copies and its resources are not necessarily busy when the primary site is alive. Our eager strategy of concentrating database log can provide long idleness to many disk drives, accordingly obtaining substantial energy saving opportunities.

6 Conclusion

The paper proposes a power-aware disaster recovery system, in which the reflection of transferred updated information can be deferred through eager compaction technique, so as to gain great energy saving of storage systems in the remote secondary site. Experiments with a commercial database system showed that 80-85% energy consumption can be conserved in the secondary-site disk storage with small penalties of possible service breakdown time.

In this paper, we focus on the energy consumption of disk drives which are main components of recent disk storage. Further, we would like to extend our approach so as to provide a system-wide analysis considering RAID controllers and cache memory.

Acknowledgement

This work has been supported in part by *Development of Out-of-Order Database Engine*, a joint research project between the University of Tokyo and Hitachi Ltd., which started in 2007 as a part of the Next-generation IT Program of the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan.

References

- [1] APC. Determining Total Cost of Ownership for Data Center and Network Room Infrastructure. White paper, 2002.
- [2] B. RUDOLPH. Storage In an age of Inconvenient Truths. Storage Network World Spring 2007, 2007.
- [3] BRITISH STANDARDS INSTITUTION. BS25999: Business Continuity Management, 2006.
- [4] CARRERA, E. V., PINHEIRO, E., AND BIANCHINI, R. Conserving Disk Energy in Network Servers. In *Proc. Int'l Conf. on Supercomputing* (2003), pp. 86–97.
- [5] COLARELLI, D., AND GRUNWALD, D. Massive Arrays of Idle Disks for Storage Archive. In *Proc. ACM/IEEE Conf. on Supercomputing* (2002), pp. 1–11.
- [6] DOUGLIS, F., KRISHNAN, P., AND BERSHAD, B. Adaptive disk spin-down policies for mobile computers. In *Proc. USENIX Symp. on Mobile and Location-Independent Computing* (1995), pp. 121–137.
- [7] EAGLE ROCK ALLIANCE. Online Survey Results: 2001 Cost of Downtime, Contingency Planning Research, 1996.
- [8] EMC CORP. Symmetrix Remote Data Facility product description guide, 2000.
- [9] F. MOORE. More power needed. Energy User News, 2002.
- [10] GNIADY, C., HU, Y. C., AND LU, Y.-H. Program Counter-Based Prediction Techniques for Dynamic Power Management. *IEEE Trans. Comput.* 55, 6 (2006), 641–658.
- [11] GOLDING, R. A., BOSCH, P., STAELIN, C., SULLIVAN, T., AND J. WILKES. Idleness is not sloth. In *Proc. USENIX Tech. Conf.* (1995), pp. 201–212.
- [12] GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., AND FRANKE, H. Reducing Disk Power Consumption in Servers with DRPM. *IEEE Computer* 36, 12 (2003), 59–66.
- [13] HEATH, T., PINHEIRO, E., HOM, J., KREMER, U., AND BIANCHINI, R. Application Transformations for Energy and Power-Aware Device Management. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Tech.* (2002), pp. 121–130.
- [14] HGST INC. Quietly cool. White Paper, HGST, 2004.
- [15] HITACHI LTD. Hitachi Relational Database Management System Solutions for Disaster Recovery to Support Business Continuity. Review Special Issue, Hitachi Technology, 2004.
- [16] IDC. The Expanding Digital Universe: A Forecast of Worldwide Information Growth Through 2010. An IDC White Paper sponsored by EMC.
- [17] JI, M., VEITCH, A., AND WIKES, J. Seneca: remote mirroring done write. In *Proc. USENIX Conf. on File and Storage Tech.* (2003), pp. 253–268.
- [18] KLEIMAN, S. Trends in Managing Data at the Petabyte Scale. Invited talk, USENIX Conf. on File and Storage Tech., 2007.
- [19] LAHIRI, T., GANESH, A., WEISS, R., AND JOSHI, A. Fast-Start: Quick Fault Recovery in Oracle. White paper, 2001.
- [20] LI, D., WANG, J., AND VARMAN, P. Conserving Energy in Conventional Disk based RAID Systems. In *Proc. Int'l Workshop on Storage Network Arch. and Parallel I/Os* (2005), pp. 65–72.
- [21] LU, H., AND LEE TAN, K. Batch Query Processing in Shared-Nothing Multiprocessors. In *Proc. Int'l Conf. on Database Syst. for Advanced Applications* (1995), pp. 238 – 245.
- [22] LU, Y., BENINI, L., AND MICHELI, G. Power-aware operating systems for interactive systems. *IEEE Trans. Very Large Scale Integration Syst.* 10, 2 (2002), 119–134.
- [23] MAXIMUM THROUGHPUT INC. Power, heat, and sledgehammer. White paper, 2002.
- [24] MEHTA, M., SOLOVIEV, V., AND DEWITT, D. J. Batch Scheduling in Parallel Database Systems. In *Proc. IEEE Int'l Conf. on Data. Eng.* (1993), pp. 400 – 410.
- [25] NATIONAL CLIMATE DATA CENTER, U.S. DOC. Climate of 2005 Atlantic Hurricane Season. Online Report available at <http://www.ncdc.noaa.gov/oa/climate/research/2005/hurricanes05.html>, 2005.
- [26] NATIONAL FIRE PROTECTION ASSOCIATION. NFPA1600: Standard on Disaster/Emergency Management and Business Continuity Programs (2004 Edition), 2004.
- [27] ORACLE CORP. Oracle Data Guard 11g. The Next Era in Data Protection and Availability. White paper, 2007.
- [28] SON, S. W., MANDEMIR, M., AND CHOUDHARY, A. Software-Directed Disk Power Management for Scientific Applications. In *Proc. IEEE Parallel and Distributed Processing Symp.* (2005), p. 4b.
- [29] U.S. SEC. General Rules and Regulations promulgated under the Securities Exchange Act of 1934, 2005.
- [30] WEDDLE, C., OLDHAM, M., QIAN, J., WANG, A. A., REIHER, P., AND KUENNING, G. PARAD: A Gear-Shifting Power-Aware RAID. In *Proc. USENIX Conf. on File and Storage Tech.* (2007), pp. 245–260.
- [31] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O – A Novel I/O Semantics for Energy-Aware Applications. In *Proc. USENIX Symp. on Operating Syst. Design and Imple.* (2002), pp. 117–130.
- [32] WORTH., S. Green Storage. SNIA Education, 2006.
- [33] YAO, X., AND WANG, J. RIMAC: A Novel Redundancy-based Hierarchical Cache Architecture for Energy Efficient, High Performance Storage System. In *Proc. EuroSys* (2006), pp. 249–262.
- [34] ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., AND WIKES, J. Hibernator: Helping Disk Arrays Sleep through the Winter. In *Proc. ACM Symp. on Operating Syst. Principles* (2004), pp. 177–190.

A Linux Implementation Validation of Track-Aligned Extents and Track-Aligned RAIDs

Jin Qian, Christopher Meyers, and An-I Andy Wang
Florida State University, {qian, meyers, awang}@cs.fsu.edu}

Abstract

Through clean-slate implementation of two storage optimizations—track-aligned extents and track-aligned RAIDs—this paper shows the values of independent validations. The experience revealed many unanticipated disk and storage data path behaviors as potential roadblocks for wide deployment of these optimizations, and also identified implementation issues to retrofit these concepts to legacy data paths.

1 Introduction

Validation studies are common in science, but less emphasized in computer science, because a rapidly moving field tends to focus on advancing the frontier.

Through a clean-slate Linux implementation of two storage optimization techniques, we aim to demonstrate the values of validations. (1) Existing validations are often implicit when the original contributors extend their work. Therefore, subtle assumptions on the OS platforms, system configurations, and hardware constraints can become obscure over time. Independent validations help identify these roadblocks, to ease the technology transfer for wide adoptions. (2) Independent validations can explore design alternatives to verify the resiliency of a concept to different platforms and hardware generations.

This paper presents a validation study of track-aligned extents [9] and track-aligned RAIDs [10]. Both showed significant performance gains. Our experience shows many unanticipated disk features and interactions along the storage data path, and identifies implementation issues to retrofit these concepts to the legacy data path.

2 Track-aligned Extents

The basic idea of track-aligned extents is that an OS typically accesses disks in blocks, each containing multiple sectors. Therefore, accessing a block can potentially cross a track boundary and incur additional head positioning time to switch tracks. By exploiting track boundaries, the performance of accessing a track size of data can improve substantially [9].

2.1 Original Implementation

Track-aligned extents [9] was built under FreeBSD by modifying FFS [7]. Two methods were proposed to extract disk track boundaries, one from the user space and one via SCSI commands. The track boundaries are extracted once, stored, and imported to FFS at

mount times. The FFS free block bitmaps are modified to exclude blocks that cross track boundaries. The FFS prefetching mechanism was modified to stop at track boundaries, so that speculative disk I/Os made for sequential accesses would respect track alignments.

Track-aligned extents rely on disks that support zero-latency access, which allows the tail-end of a requested track to be accessed before the beginning of the requested track content [13]. This feature allows an aligned track of data to be transferred without rotational overhead.

With Quantum Atlas 10K II disks, the measured results showed 50% improvement in read efficiency. Simulated and computed results also demonstrated improved disk response times and support for 56% higher concurrency under video-on-demand workloads.

2.2 Recreating Track-aligned Extents

Recreating track-aligned extents involves (1) finding the track boundaries and the zero-latency access disk characteristics, (2) making use of such information, and (3) verifying its benefits. The hardware and software experimental settings are summarized in Table 1.

Hardware/software	Configurations
Processor	Pentium D 830, 3GHz, 16KB L1 cache, 2x1MB L2 cache
Memory	128 MB or 2GB
RAID controller	Adaptec 4805SAS
Disks tested	Maxtor SCSI 10K5 Atlas, 73GB, 10K RPM, 8MB on-disk cache [6] Seagate CheetahR 15K.4 Ultra320 SCSI, 36GB, 8MB on-disk cache [12] Fujitsu MAP3367NC, 10K RPM, 37GB, with 8MB on-disk cache [5]
Operating system	Linux 2.6.16.9
File system	Ext2 [4]

Table 1: Experimental settings.

2.3 Extracting Disk Characteristics

User-level scanning: Since the reported performance gains for track alignments are high, conceivably a user-level program can observe timing variations to identify track boundaries. A program can incrementally issue reads, requesting one more sector than before, starting from the 0th sector. As the request size grows, the disk bandwidth should first increase and then drop as the request size exceeds the size of the first track (due to track switching overhead). The process can then repeat, starting from the first sector of the previously found track. Binary search can improve the scheme.

To reduce disturbances caused by various disk data path components, we used the `DIRECT_IO` flag to

bypass the Linux page cache, and we accessed the disk as a raw device to bypass the file system. We used a modified `aacraid` driver code to bypass the SCSI controller, and we used `sdparm` to disable the read cache (`RCD=1`) and prefetch (`DPTL=0`) of the disk.

As a sanity check, we repeated this experiment with an arbitrary starting position at the 256th sector (instead of the 0th sector). Additionally, we repeated this experiment with a random starting sector between 0 and 512, with each succeeding request size increasing by 1 sector (512 bytes).

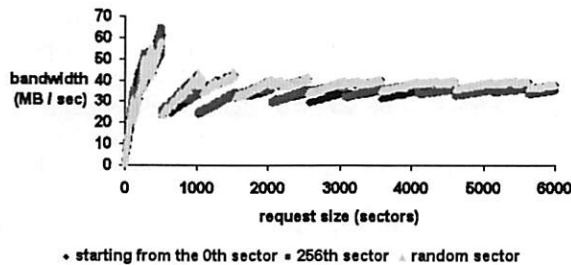


Figure 1: Bandwidth for various read request sizes from varying starting sectors on a Maxtor disk.

Surprisingly, although Figure 1 exhibits bandwidth “cliffs,” the characteristic trends are not sensitive to the starting location of requests, suggesting that those cliffs are caused by sources of data misalignments other than tracks. Some possibilities are transfer granularity of the DMA and the management granularity of IO buffers. The graph also suggests the presence of other optimizations that are not disabled. For example, the high bandwidth before the first cliff far exceeds our expected performance gain. [2] conjectures that the DEC prefetch scheme implemented in Maxtor may override specified disk settings at times and proceed with prefetching. Additionally, for certain ranges of request sizes (e.g., between 1,000 and 1,500 sectors), the average bandwidth shows multimodal behaviors.

To verify that those cliff locations are not track boundaries, we wrote a program to access random cliff locations with the access size of 512 sectors (256KB), as indicated by the first cliff location. We ran multiple instances of this program concurrently and perceived no noticeable performance difference compared to the cases where the accesses started with random sectors.

SCSI diagnostic commands: Unable to extract track boundaries from a naive user-level program, we resorted to SCSI `SEND/RECEIVE DIAGNOSTIC` commands to map a logical block address (LBA) to a physical track, surface, and sector number.¹ However, this translation for large drives is very slow, and it took days to analyze a 73-GB drive. We modified the `sg_senddiag` program in the Linux `sg3_utils` package to speed up the extraction process, according to the following pseudocode:

¹ We did not use DIXtrac [8] for the purpose of clean-slate implementation and validation.

1. Extract from LBA 0 sector-by-sector until either track number or surface number changes. Record LBA and the physical address of this track boundary. Store the track size S.
2. Add S to the last known track boundary T and translate $S + T$ and $S + T - 1$.
 - a. If we detect a track change between $S + T$ and $S + T - 1$, then $S + T$ is a new boundary. Record the boundary. Go to step 2.
 - b. If there is no change between $S + T$ and $S + T - 1$, the track size has changed. Extract sector-by-sector from the previous boundary until we detect a new track boundary. Record the boundary, update S, and go to step 2.
3. If sector reaches the end of the disk in step 2, exit.

Through this scheme, we extracted the layout mapping specifics that are not always published in vendors’ datasheets and manuals [5, 6, 12] in about 7 minutes.

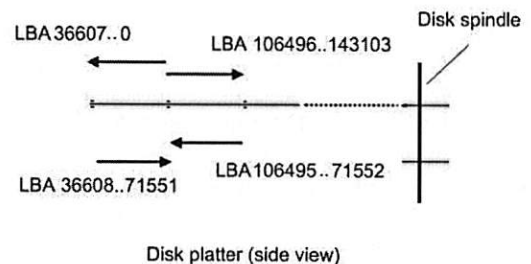


Figure 2: Non-monotonic mapping between LBA and track numbers.

First, the LBA mapping to the physical track number is not monotonic (Figure 2). For the Maxtor drive, LBA 0 starts on track 31 of the top surface and increases outward (from the disk spindle) to track 0, and then the LBA continues from the bottom surface of track 0 inward to track 31. Next, the LBA jumps to track 63 of the bottom surface growing outward to track 32, and then switches back to the top surface’s track 32 and continues inward to track 63. The pattern repeats.

Variants of this serpentine numbering scheme [1, 11] are observed in Seagate [12] and Fujitsu [5] drives as well. At the first glance, one might conjecture this numbering scheme relates to the elevator and scanning-based IO schedulers, but this scheme is attributed to the faster timing when switching a head track-to-track on the same surface than when switching to a head on a different surface [11].

Second, the track size differs even for the same disk model from the same vendor, due to the manufacturing process of the disks. After assembly, the disk head frequency response is tested. Disk heads with a lower frequency response are formatted with fewer sectors per track [2]. We purchased 6 Maxtor 10K V drives at the same time and found 4 different LBA numbering schemes (Table 2). The implication is that track extraction needs to be performed on every

disk, even those from the same model. Track size may differ in the same zone on the same surface due to defects. Thus, we are no longer able to calculate the track boundary with zone information but have to extract all tracks.

Serial number	Surface 0, outer most track	Surface 1, outer most track
J20 Q3 CZK	1144 sectors	1092 sectors
J20 Q3 C0K/J20 Q3 C9K	1092 sectors	1144 sectors
J20 TK 7GK	1025 sectors	1196 sectors
J20 TF S0K/J20 TF MKK	1060 sectors	1170 sectors

Table 2: Track sizes of Maxtor 10K V drives.

Verifying track boundaries: To verify track boundaries, we wrote a program to measure the elapsed time to access 64 sectors with shifting offsets from random track boundaries. The use of 64 sectors eases the visual identifications of boundaries. We measured tracks only from the top surface within the first zone of a Maxtor disk, so we could simplify our experiment by accessing a mostly uniform track size of 1,144 sectors.

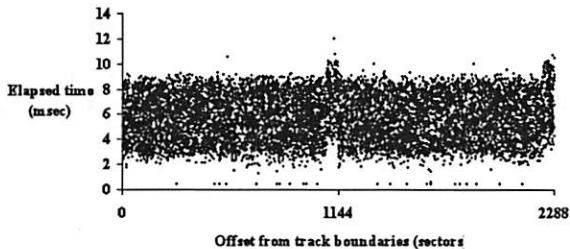


Figure 3: Elapsed time to access 64 sectors, starting from different offsets from various track boundaries on a Maxtor drive (the track size is 1,144 sectors).

Figure 3 confirms extracted track boundaries. Each data point represents the time to access a 64-sector request starting from a randomly chosen sector offset from a track boundary. The 6-msec timing variation reflects the rotation delay for a 10,000 RPM drive. The average elapsed time for accessing 64 sectors across a track boundary is 7.3 msec, compared to 5.7 msec for not crossing the track boundaries. Interestingly, the difference of 1.6 msec is much higher than the track switching time of 0.3 to 0.5 msec [6]. We also verified this extraction method with other vendor drives. The findings were largely consistent.

Zero-latency feature verification: Since the effectiveness of track-aligned extents relies on whether a disk can access the data within a track out-of-order, we performed the tests suggested in [13]. Basically, we randomly picked two consecutive sectors, read those sectors in reverse LBA order, and observed the timing characteristics. We performed the test with various caching options on.

As shown in Figure 4, with a Maxtor drive, 50% of the time the second request is served from the on-disk cache, indicating the zero-latency capability. (We did not observe correlations between the chosen sectors and

whether the zero-latency feature is triggered.) In contrast, the other two drives always need to wait for a 3- to 6-msec rotational delay before serving the second sector request. For the remainder of the paper, we will use the Maxtor drives.

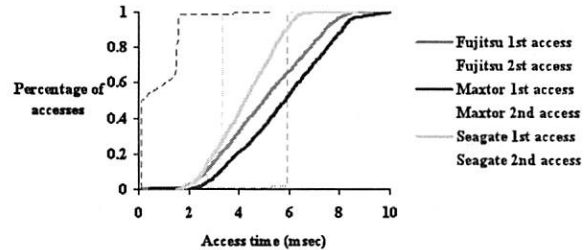


Figure 4: CDF of disk access times for accessing random sets of two consecutive LBAs in the reverse order.

2.4 Exploiting Track Boundaries

The track boundary information can be exploited at different levels.

User level: One possibility is to create a user program to make use of this track information. Similar to the disk defragmentation, instead of moving file blocks to reduce the level of fragmentation, we can move blocks to align with track boundaries. This approach avoids kernel changes and can make files smaller than a track not cross track boundaries, and files larger than a track aligned to track boundaries.

However, this approach needs to overcome many tricky design points. For example, certain blocks are referenced from many places (e.g., hardlinks). Moving those blocks requires tracking down and updating all references to the block being moved. Such information might not be readily available.

File system level: We can mark certain sectors as bad so a file system cannot allocate blocks that consist of sectors across track boundaries. However, this method does not prevent a track-size file from being allocated across two tracks. This approach also anticipates some bandwidth loss when a single IO stream accesses multi-track files due to unused sectors. However, when a system is under multiple concurrent IO streams, the performance benefits of accessing fewer tracks when multiplexing among streams can outweigh the performance loss.

Implementation: We implemented track-aligned extents in ext2 [4] under Linux. First, we used the track boundary list extracted by the SCSI diagnostic commands as the bad-block list input for the `mke2fs` program, which marks all of these blocks, so that they will not be allocated to files. We also put this list in a kernel module along with two functions. One initializes and reads the list from user space. The other is used by different kernel components to find a track boundary after a given position.

We then modified the ext2 pre-allocation routine to allocate in tracks (or up to a track boundary). One disadvantage of this approach is over-allocation, but the unused space can later be returned to the system. However, should the system anticipate mostly track-size accesses, we are less concerned with the wasted space. For instance, database and multimedia applications can adjust their access granularity accordingly. With the aid of this list, we can also change the read-ahead to perform prefetches with respect to track boundaries.

Our experience suggests that individual file systems only need to make minor changes to benefit from track alignments.

2.5 Verification of the Performance Benefits

We used the sequential read and write phases of the Bonnie benchmark [3], which is unaware of the track alignments. The write phase creates a 1-GB file, which exceeds our 128-MB memory limit. We enabled SCSI cache, disk caching, and prefetch to reflect normal usage. Each experiment was repeated 10 times, analyzed at a 90% confidence interval.

Figure 5 shows the expected 3% slowdown for a single stream of sequential disk accesses, where skipped blocks that cross track boundaries can no longer contribute to the bandwidth.

We also ran *diff* from GNU *diffutils* 2.8.1 to compare two 512-MB large files via interleaved reads between two files, with the *-speed-large-files* option. Without this option, *diff* will try to read one entire file into the memory and then the other file and compare them if memory permits, which nullifies our intent of testing interleaved reads. Figure 6 shows that track-aligned accesses are almost twice as fast as the normal case. In addition, we observed that disk firmware prefetch has no regard for track boundaries. Disabling on-disk prefetch further speeds up track-aligned access by another 8%. Therefore, for subsequent experiments, we disabled disk firmware prefetch for track-aligned accesses.

Additionally, we conducted an experiment that involves concurrent processes issuing multimedia-like traffic streams at around 500KB/sec. We used 2GB for our memory size. We wrote a script that increases the number of streams by one after each second, and the script records the startup latency of each new stream. Each emulated multimedia streaming process first randomly selects a disk position and sequentially accesses the subsequent blocks at the specified streaming rate. We assumed that the acceptable startup latency is around 3 seconds, and the program terminates once the latency reaches 3 seconds.

Figure 7 shows that the original disk can support up to 130 streams with a startup latency within 3 seconds. A track-size readahead window can reduce the latency at 130 streams by 30%, while track-aligned access can reduce the latency by 55%.

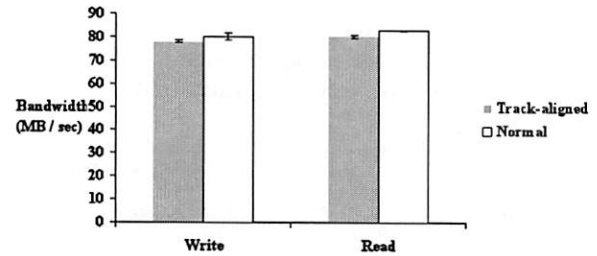


Figure 5: Bandwidth comparisons between conventional and track-aligned accesses to a single disk, when running the Bonnie benchmark.

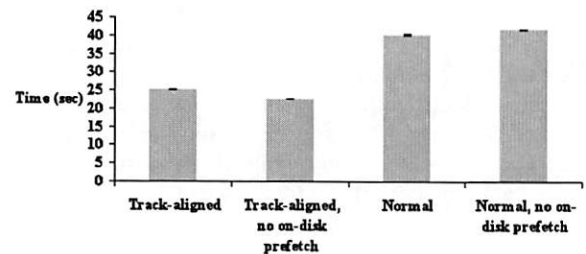


Figure 6: Speed comparisons between conventional and track-aligned accesses to a single disk, diffing two 512MB files with 128MB of RAM.

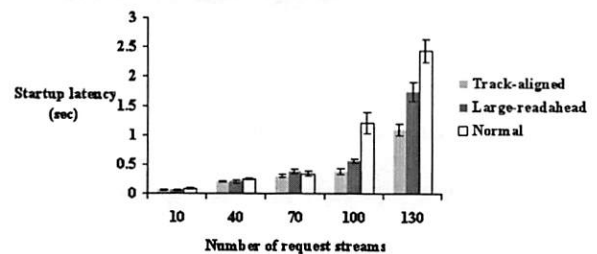


Figure 7: Startup latency comparisons of conventional I/O requests, requests with a one-track prefetch window, and track-aligned requests on a single disk, with a varying number of multimedia-like request streams.

3 Track-aligned RAIDs

Original implementation: Schindler et al [10] proposed Atropos, a track-aligned RAID. The implementation was through a user-level logical volume manager process. The process bypasses conventional storage data paths and issues raw IOs. An application needs to be linked with a stub library to issue reads and writes. The library uses shared memory to avoid data copies and communicates with Atropos through a socket.

Without the conventional storage data path, Atropos is responsible for scheduling requests with the help of a detailed disk model. Atropos also needs to duplicate logics provided by conventional RAID levels. As a proof of concept, the measured prototype implemented RAID-0 (no redundancy) and RAID-1

(mirroring), although issues relevant to other RAID levels are addressed in the design.

To handle different track sizes due to disk defects, for simplicity Atropos skips tracks that contain more than a threshold number of defects, which translates to about 5% of storage overhead.

The performance for track-aligned RAID5 matches the efficiency expectation of track-aligned extents.

Recreating Track-aligned RAID5: Our clean-slate validation implements track-aligned RAID5 via modifying RAID-5 (distributed parity), retrofitting the conventional storage data path. Thus, unmodified applications can enjoy the performance benefit as well. However, we had to overcome a number of implementation constraints.

Recall from Section 2.3 that the track sizes can differ even from the same disk model. This difference was much more than that caused by defects. Therefore, we need measures beyond skipping tracks. For one, we can construct stripes with tracks of different sizes. Although this scheme can work with RAID-0, it does not balance load well or work well with other RAID levels. For example, RAID-5 parity is generated via XORing chunks (units of data striping) of the same size. Suppose we want the chunk unit to be set to the size of a track. If we use the largest track size as the chunk unit, some disks need to use 1+ tracks to form a chunk. Or we can use the smallest track size as the chunk unit, leading to about 10% of unused sectors for disks with larger track sizes.

Additionally, we observed that parity in RAID5 can interact poorly with prefetching in the following way. Take RAID-5 as an example. At the file system level, prefetching one track from each non-parity disk involves a prefetching window that is the size of a track multiplied by the number of disks that do not contain the parity information. However, as a RAID redirects the contiguous prefetching requests from the file system level, the actual forwarded track-size prefetching requests to individual disks are fragmented, since reads in RAID5 do not need to access the parity information.

Another poor interaction is the Linux plug and unplug mechanisms associated with disk queues and multi-device queues. These mechanisms are designed to increase the opportunities for data reordering by introducing artificial forwarding delays at times (e.g., 3 msec), and do not respect track boundaries. Therefore, by making these mechanisms aware of track boundaries, we were finally able to make individual disks in a RAID-5 access in a track-aligned manner.

Implementation: We modified Linux software RAID-5 to implement the track-aligned accesses. We altered the `make_request` function, which is responsible for translating the RAID virtual disk address into individual disk addresses. If the translated requests crossed track boundaries, the unplug functions for individual disk queues were explicitly invoked to issue track-aligned requests.

To prevent the parity mechanisms from fragmenting track-size prefetching requests, we modified RAID-5. Whenever the parity holding disk in a stripe was the only one not requested for that stripe, we filled in the read request for that disk and passed it down with all others. When this dummy request was completed, we simply discarded the data. The data buffer in Linux software RAID-5 is pre-allocated at initialization, so this implementation does not cause additional memory overhead.

Verification of performance benefits: We compared the base case RAID-5 with a track-aligned RAID-5 with five disks, and a chunk size of 4KB. For the Bonnie benchmark, we used a 1-GB working set with 128MB of RAM. Figure 8 shows that the write bandwidth for the three system settings falls within a similar range due to buffered writes. However, for read bandwidth, the track-aligned RAID-5 outperforms the conventional one by 57%.

The `diff` experiment compared two 512-MB files with 128MB of RAM. Figure 9 shows that the track-aligned RAID-5 can achieve a 3x factor speedup compared to the original RAID-5.

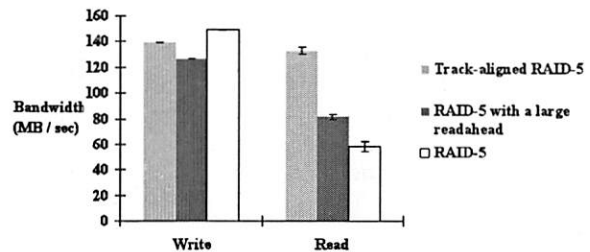


Figure 8: Bandwidth comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of four tracks, and the original RAID-5, running Bonnie with 1GB working set and 128MB of RAM.

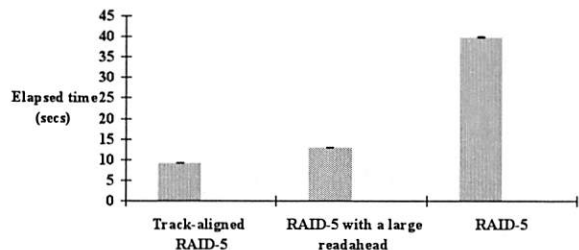


Figure 9: Elapsed time comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of four tracks, and the original RAID-5, when running `diff` comparing two 512MB files.

For the multimedia-like workload with 2GB of RAM, the track-aligned RAID-5 demonstrates a 3.3x better scaling in concurrency than the conventional RAID-5 (Figure 10), where a RAID-5 with a readahead window comparable to the track-aligned RAID-5 contributes only less than half of the scaling improvement. The latency improvement of track-aligned RAID-5 is

impressive considering that the RAID-5 was expected to degrade in latency when compared to the single-disk case, due to the need to wait for the slowest disk for striped requests. Track-aligned accesses reduce the worst-case rotational timing variance and can realize more benefits of parallelism.

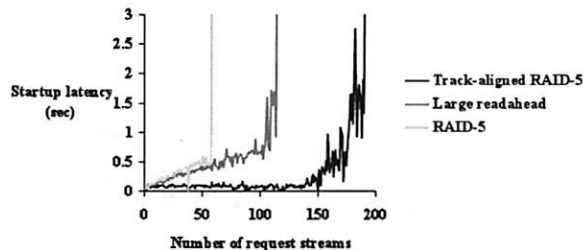


Figure 10: Startup latency comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of four tracks, and the original RAID-5, with a varying number of multimedia-like request streams.

4 Lessons Learned and Conclusions

Through clean-slate implementations of track-aligned extents and track-aligned RAID5s, we have demonstrated important values of independent validations. First, the validation of research results obtained five years ago shows the relative resiliency and applicability of these concepts to different platforms and generations of disks. On the other hand, as the behaviors of disks and the legacy storage data path become increasingly complex, extracting physical disk geometries will likely become increasingly more difficult. Also, as disks become less homogeneous even within the same model, techniques such as track-aligned RAID5s need to devise additional measures to prevent a RAID from being limited by the slowest disk.

Second, through exploring design and implementation alternatives, we revealed many unanticipated interactions among layers of data path optimizations. On-disk prefetching, IO scheduling and aggregation, RAID parity, file system allocation, and file system prefetching—all have side effects on IO access alignment and profound performance implications. Unfortunately, the interfaces among data path layers are lacking in expressiveness and control, leading to modifications of many locations to retrofit the concepts of access alignment into the legacy storage data path, the remedy for which is another fruitful area of research to explore.

Acknowledgements

We thank Mark Stanovich and Adaptec for helping us bypass some RAID controller features. We also thank Peter Reiher and Geoff Kuenning for reviewing this paper. This research is sponsored by NSF CNS-

0410896 and CNS-0509131. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the NSF, FSU, or the U.S. government.

References

- [1] Anderson D. You Don't Know Jack about Disks. *Storage*. 1(4), 2003.
- [2] Anonymous Reviewer, reviewer comments, *the 6th USENIX Conf. on File and Storage Technologies*, 2007.
- [3] Bray T. Bonnie benchmark. <http://www.textuality.com/bonnie/download.html>, 1996.
- [4] Card R, Ts'o T, Tweedie S. Design and Implementation of the Second Extended Filesystem. *The HyperNews Linux KHG Discussion*. <http://www.linuxdoc.org>, 1999.
- [5] Fujitsu MAP3147NC/NP MAP3735NC/NP MAP3367NC/NP Disk Drives Product/Maintenance Manual. http://www.fujitsu.com/downloads/COMP/fcpa/hdd/discontinued/map-10k-rpm_prod-manual.pdf, 2007.
- [6] Maxtor Atlas 10K V Ultra320 SCSI Hard Drive. <http://www.darklab.rutgers.edu/MERCURY/t15/disk.pdf>, 2004.
- [7] McKusick MK, Joy WN, Leffler SJ, Fabry RS. A Fast File System for UNIX, *Computer Systems*, 2(3), pp. 181-197, 1984.
- [8] Schindler J, Ganger GR. Automated Disk Drive Characterization. CMU SCS Technical Report CMU-CS-99-176, December 1999.
- [9] Schindler J, Griffin JL, Lumb CR, Ganger GR. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. *Proc. of the 1st USENIX Conf. on File and Storage Technologies*, 2002.
- [10] Schindler J, Schlosser SW, Shao M, Ailamaki A, Ganger GR. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. *Proc. of the 3rd USENIX Conf. on File and Storage Technologies*, 2004.
- [11] Schlosser SW, Schindler J, Papadomanolakis S, Shao M, Ailamaki A, Faloutsos C, Ganger GR. On Multidimensional Data and Modern Disks. *Proc. of the 4th USENIX Conf. on File and Storage Technology*, 2005.
- [12] Seagate Product Manual: CheetahR 15K.4 SCSI. <http://www.seagate.com/staticfiles/support/disc/manual/s/enterprise/cheetah/15K.4/SCSI/100220456d.pdf>, 2007.
- [13] Worthington BL, Ganger GR, Patt YN, Wilkes J. On-line Extraction of SCSI Disk Drive Parameters. *ACM Sigmetrics*, 1

Automatic Optimization of Parallel Dataflow Programs

Christopher Olston, Benjamin Reed, Adam Silberstein, Utkarsh Srivastava

Yahoo! Research

{olston, breed, silberst, utkarsh}@yahoo-inc.com

Abstract

Large-scale parallel dataflow systems, e.g., Dryad and Map-Reduce, have attracted significant attention recently. High-level dataflow languages such as Pig Latin and Sawzall are being layered on top of these systems, to enable faster program development and more maintainable code. These languages engender greater transparency in program structure, and open up opportunities for automatic optimization. This paper proposes a set of optimization strategies for this context, drawing on and extending techniques from the database community.

1 Introduction

There is significant recent interest in parallel dataflow systems and programming models, e.g., Dryad [19], Jaql [17], Map-Reduce [9], Pig [22] and Sawzall [23]. While the roots of this work date back several decades in the database, programming language and systems communities, the emergence of new and well-funded application areas requiring very large-scale parallel processing is driving this work in somewhat different directions than in the past. In particular, recent work concentrates on much larger-scale systems, simpler fault-tolerance and consistency mechanisms, and stylistically different languages. This work is leading to a new computing paradigm, which some have dubbed *Data-Intensive Scalable Computing* (DISC)¹ [8].

The DISC world is being built bottom-up. Google's Map-Reduce [9] system introduced scalable, fault-tolerant implementations of two key dataflow primitives: independent processing of (groups of) records, and agglomeration of records that contain matching values. Then came Dryad [19], with built-in support for general dataflow graphs, including operator chains of arbitrary length and in- and out-bound branching. Now higher-level languages are being layered on top of these systems, to translate abstract user-supplied dataflow or query expressions into underlying parallel dataflow graphs, e.g., DryadLINQ [21], Jaql [17], Pig Latin [22] and Sawzall [23].

These high-level languages engender a relatively high degree of transparency in program structure. For example, standard data manipulation operations such as *join*

and *filter* are expressed via declarative primitives. Also, multistep processing is explicitly broken down into the constituent steps, e.g., *join* followed by *filter* followed by *face recognition*, rather than being buried inside low-level constructs such as Map functions.

These forms of transparency, in addition to making programs easier to write, understand and maintain, also open up opportunities for automatic optimization. This paper proposes a set of optimization strategies for the DISC paradigm. The aim is not to provide concrete or proven results, but rather to suggest some jumping-off points for work in this area. Our belief is that good optimization technology, combined with the economics of programmer resources becoming more expensive relative to computer resources, will trigger a mass migration from low-level programming (e.g., direct Map-Reduce or Dryad programs) to high-level programming (e.g., Jaql, Pig Latin, SQL), analogous to the migration from assembly to C-style languages to Java-style languages.

1.1 Background: Pig

This paper is motivated by our experience with *Pig*, an open-source [4] dataflow engine used extensively at Yahoo! to process large data sets. Pig compiles Pig Latin programs, which are abstract dataflow expressions, into one or more physical dataflow jobs, and then orchestrates the execution of these jobs. Pig currently uses the Hadoop [3] open-source Map-Reduce implementation as its physical dataflow engine.

The current Pig implementation incorporates two simple but critical optimizations: (1) Pig automatically forms efficient pipelines out of sequences of per-record processing steps. (2) Pig exploits the *distributive* and *algebraic* [15] properties of certain aggregation functions, such as COUNT, SUM, AVERAGE and some user-defined functions, and automatically performs partial aggregation early (known as *combining* in the Map-Reduce framework), to reduce data sizes prior to the expensive data partitioning operation.

Our users are demanding more aggressive optimization of their programs, and indeed there is room to do much more on the optimization front.

¹Previously "Data-Intensive Supercomputing."

1.2 DISC vs. DBMS Optimization

In developing automatic optimization techniques for DISC, we can draw upon many ideas and techniques from the database community, which has studied set-oriented data processing for decades, including parallel processing on clusters [7, 11, 13]. While DISC systems bear a strong resemblance to parallel query processors in database management systems (DBMS), the context is somewhat different: The DBMS context emphasizes highly declarative languages, normalized data and strong consistency, whereas DISC is geared toward procedural code, flexible data models, and cost-effective scalability through weak consistency and commodity hardware.

Traditional DBMS optimization techniques [18] are *model-based*, i.e., they search for “optimal” execution strategies over models of the data, operators and execution environment. In the DISC context, accurate models may not be available a priori, because: (1) Data resides in plain files for ease of interoperability with other tools, and the user may not instruct the system how to parse the data until the last minute; (2) Many of the operators are governed by custom user-supplied code whose cost and data reduction/blowup properties are not known a priori; (3) DISC uses large pools of unreliable and perhaps heterogeneous machines, and formulating simple and accurate models of the execution environment is a challenge.

Starting around 2000, motivated in part by considerations related to the ones stated above, the database community has begun studying *adaptive* approaches to one optimization problem: query plan selection [6, 10]. Adaptive query planning does not rely on the a-priori existence of accurate models, and instead adopts a trial-and-error, feedback-driven approach.

1.3 Model-Light Approach

DISC provides an interesting opportunity to revisit many of the specifics of query optimization in a new light. In particular, it makes sense to pursue a *model-light* approach, guided by the following principles:

1. **Discriminative use of information.** Optimization decisions should not be made on the basis of unreliable information. Conversely, information known to be reliable should be leveraged. For example, reliable file size metadata can typically be obtained, and knowing file sizes can be useful in selecting join algorithms (Section 3.2) and scheduling overlapping programs (Section 4.1). As another example, although the system may not have reliable cost and selectivity estimates for all operations, certain ones such as projection, simple filtering and counting are known to be cheap and data-reducing, and hence ought to be placed early in the execution sequence when possible (Section 3.1).

2. **Risk avoidance.** In cases where key optimization parameters are missing or unreliable, the optimization process should be geared toward minimizing the risk of a bad outcome. For example, when selecting derived data to cache, in the absence of reliable models for the size and utility of various derived data sets, the system should construct a diverse portfolio of cached content (Section 4.2.2). This strategy is less risky than betting on one particular category of derived data being the most useful, according to an unreliable model.
3. **Adaptivity.** Key parameters like intermediate data sizes and black-box function costs, which are hard to estimate a priori, can be measured at runtime. Based on measurements taken at runtime, the system may be able to adjust its execution and storage tactics on the fly, to converge to a better strategy over time. Aspects that are, at least in principle, amenable to adaptive optimization at runtime include dataflow graph structure (see [6, 10, 16]), load balancing across partitions (see [5, 26]), data placement (Section 4.2.1), and caching and reuse of derived data (Section 4.2.2).

In this paper we lay out some possible optimization strategies for DISC that align with the above principles. Section 3 focuses on *single-program optimizations* that optimize one program at a time, and highlights ideas from the database community that may be applicable in the DISC context. Section 4 focuses on *cross-program optimizations* that amortize IO and CPU work across related programs, and proposes several novel approaches.

Before discussing these optimizations we briefly describe our Pig Latin language, as a concrete example of a dataflow language that can be optimized.

2 Pig Latin Language Overview

Pig Latin is our high-level dataflow language for expressing computations or transformations over data. We illustrate the salient features of Pig Latin through an example.

Example 1 Suppose we have search query logs for May 2005 and June 2005, and we wish to find “add-on terms” that spike in June relative to May. An *add-on term* is a term that is used in conjunction with standard query phrases, e.g., due to media coverage of the 2012 Olympics selection process there may be a spike in queries like “New York olympics” and “London olympics,” with “olympics” being the add-on term. Similarly, the term “scientology” may suddenly co-occur with “Tom Cruise,” “depression treatment,” and other phrases. The following Pig Latin program describes a dataflow for identifying June add-on terms (the details of the syntax are not important for this paper).²

²In reality, additional steps would be needed to eliminate pre-existing add-ons like “City” in “New York City”; due to space constraints we leave the additional steps as an exercise for the reader.

```

# load and clean May search logs
1. M = load '/logs/may05' as (user, query, time);
2. M = filter M by not isURL(query);
3. M = filter M by not isBot(user);

# determine frequent queries in May
4. M_groups = group M by query;
5. M_frequent = filter M_groups by COUNT(M) > 10^4;

# load and clean June search logs
6. J = load '/logs/june05' as (user, query, time);
7. J = filter J by not isURL(query);
8. J = filter J by not isBot(user);

# determine June add-ons to May frequent queries
9. J_sub = foreach J generate query,
    flatten(Subphrases(query)) as subphr;
10. eureka = join J_sub by subphr,
    M_frequent by query;
11. addons = foreach eureka generate
    Residual(J_sub::query, J_sub::subphr) as residual;

# count add-on occurrences, and filter by count
12. addon_groups = group addons by residual;
13. counts = foreach addon_groups generate residual,
    COUNT(addons) as count;
14. frequent_addons = filter counts by count > 10^5;
15. store frequent_addons into 'myoutput.txt';

```

Line 1 specifies the filename and schema of the May query log. Lines 2 and 3 filter out search queries that consist of URLs or are made by suspected “bots” (the filters are governed by the custom Boolean functions `isURL` and `isBot`, which have been manually ordered to optimize performance). Lines 4–5 identify frequent queries in May.

Lines 6–8 are identical to Lines 1–3, but for the June log. Lines 9–10 match sub-phrases in the June log (enumerated via a custom set-valued function `Subphrases`) against frequent May queries. Line 11 then extracts the add-on portion of the query using a custom function `Residual` (e.g., “olympics” is an add-on to the frequent May query “New York”).

Lines 12–14 count the number of occurrences of each add-on term, and filter out add-ons that did not occur frequently. Line 15 specifies that the output should be written to a file called `myoutput.txt`.

In general, Pig Latin programs express acyclic dataflows in a step-by-step fashion using variable assignments (the variables on the left-hand side denote sets of records). Each step performs one of: (1) data input or output, e.g., Lines 1, 6, 15; (2) relational-algebra-style transformations, e.g., Lines 10, 14; (3) custom processing, e.g., Lines 9, 11, governed by *user-defined functions* (UDFs). A complete description of the language is omitted here due to space constraints; see [22].

3 Single-Program Optimizations

Optimization of a single dataflow program can occur in two stages:

- **Logical** optimizations restructure the logical dataflow graph supplied by the user. These optimizations produce a new graph that is semantically equivalent to

the original one but implies a more efficient evaluation strategy. An example is to move a cheap filter ahead of more expensive operations with which the filter commutes.

- **Physical** optimizations pertain to how the logical dataflow graph is converted into a physical execution plan, e.g., as a sequence of Map-Reduce jobs. An example is to encode a sequence of two filters as a single Map operation.

3.1 Logical Optimizations

Examples of textbook [24] logical optimizations that are consistent with our model-light optimization philosophy (Section 1.3) include:

Early projection. Projection refers to the process of retaining only a subset of the fields of each record, and discarding the rest. A well-known optimization is to project out unneeded fields as early as possible.

Early filtering. Entire unneeded records can be eliminated early via filtering. In view of our discriminative information use and risk avoidance tenets, a filter should only be moved to an earlier position if there is enough confidence that it is cheap relative to its data reduction power. Thus, if the filter involves a UDF, it is better to leave the filter at the user-specified position since the UDF might be expensive and moving it earlier may cause it to be invoked more times than the user intended. As a possible extension, the cost and data-reducing power of UDFs can be discovered and exploited on the fly, using adaptive query processing techniques [10].

Operator rewrites. Sometimes, certain operator sequences can be recognized and converted into equivalent operators that have much more efficient implementations. For example, if a user writes a cross product of two data sets followed by a filter on the equality of two attributes, the cross and filter operators can be collapsed into a join operator that can typically be implemented more efficiently than a cross product.

3.2 Physical Optimizations

In the current Pig implementation, each logical dataflow graph is compiled into a sequence of Map-Reduce [9] jobs. A Map-Reduce job consists of five processing stages:

1. *Map*: process fragments of the input data set(s) independently, and assign *reduce keys* to outgoing data items.
2. *Combine* (optional): process all data items output by a given map instance that have the same reduce key, as a unit.
3. *Shuffle*: transmit all data items with a given reduce key to a single location.

4. *Sort*: sort all items received at a location, with the reduce key as a prefix of the sort specification.
5. *Reduce*: process all data items that have the same reduce key as a unit.

Logical operations need to be mapped into these stages. Sometimes an efficient mapping requires splitting a single logical operation into multiple physical ones. For example, a logical duplicate elimination operation that is to occur in the reduce stage can be converted into a sort operation (which can be incorporated into the Map-Reduce sort stage) followed by an operator that eliminates adjacent duplicate data items in a streaming fashion, in the reduce stage.

Join is perhaps the most critical operation, because it can be very expensive (quadratic cost in the worst case). There are several alternative ways to translate a logical join operation into a physical execution plan [14], including:

- **Symmetric join**: Pass both data sets through the same shuffle-sort sequence, using the join attribute(s) as the reduce key. Then match pairs of joining records in the reduce stage.
- **Symmetric join over prepartitioned data**: If the data is already partitioned on (a subset of) the join attribute(s), do the matching in the map stage, at which point the join is complete. (The shuffle, sort and reduce stages are not used.)
- **Asymmetric join**: Perform a map stage over fragments of the larger data set, and in each map instance read a full copy of the smaller data set to perform matching. (This execution method avoids having to shuffle the larger data set.)

The optimal join execution strategy in a given situation depends on whether the data is prepartitioned, and on the sizes of the input data sets (asymmetric join can be best if one data set is very small relative to the other). The cost differences among strategies can span orders of magnitude, so it is important to choose wisely.

If the join occurs early in the dataflow and processes data directly out of stored files, then the choice of join method can be driven by basic system metadata such as file sizes and file partitioning method (if any). If the join occurs late in the dataflow, following operators whose data reduction/blowup behavior has not been modeled well, then there is less hope of selecting a good join strategy in advance. In the latter case, an adaptive “wait and see” approach may make sense, even though doing so may incur additional materialization overhead and/or wasted work due to aborted trials.

4 Cross-Program Optimizations

We now consider combined optimization of multiple dataflow programs, perhaps submitted independently by different users. This type of optimization is of interest if the number of (popular) data sets is much less than the number of users processing those data sets. At internet companies like Yahoo!, hundreds of users pour over a handful of high-value data sets, such as the web crawl and the search query log. In data-intensive environments, disk and network IO represent substantial if not dominant execution cost components, and it is desirable to amortize these costs across multiple programs that access the same data set.

In some cases, programs that access the same data set also perform redundant computation steps. Dataflow programs tend to propagate among users via a cut-paste-modify model whereby users pass around code fragments over email lists or forums. In some cases users explicitly link their dataflow graphs to subgraphs published by other users, using tools like Yahoo! Pipes [27]. Hence it is often the case that programs submitted by different users exhibit a common prefix. For example, a processing prefix that occurs frequently at Yahoo! is: (1) access the web crawl, (2) remove spam pages, (3) remove foreign-language pages, (4) group pages by host for subsequent host-wise processing.

Mechanisms to amortize work across related programs fall into two categories: *concurrent* and *nonconcurrent* work sharing. Concurrent work sharing entails executing related programs in a joint fashion so as to perform common work only once. Nonconcurrent sharing entails caching the result of IO or CPU work performed while evaluating one program, and leveraging it for future programs. We discuss each category in turn.

4.1 Concurrent Work Sharing

If a set of programs sharing a common prefix or subexpression are encountered in the system’s work queue at the same time, an optimizing compiler can create a single branching dataflow for the simultaneous evaluation of all the programs [25]. DISC workloads are dominated by IO-bound programs that scan large data files from end to end. In this context the most important “shared prefix” is the scan of the input file(s). Techniques exist for sharing file scans among concurrent programs [12].

In the presence of opportunities and mechanisms to share work among concurrent programs, the key open question is how to *schedule* programs to maximize the sharing benefit. In particular:

1. Given that coupling a slow program with a fast one may increase the response time of the latter, under what conditions should the system couple them?

2. Under what conditions is it beneficial to defer execution of an enqueued program in anticipation of receiving future sharable programs?

Typically, DISC systems do not attempt to guarantee a fast response time for any *individual* program. Rather, the aim is to make efficient use of resources and achieve low *average* response time. Hence we address the two questions above in the context of minimizing average response time. We begin with Question 1.

Suppose a pair of sharable programs P_1 and P_2 have individual execution times t_1 and t_2 , respectively, but incur a lesser total time $t_{1+2} < t_1 + t_2$ if merged and evaluated jointly. Let t^s denote the sharable component of these programs, such that $t_{1+2} = t^s + t_1^n + t_2^n$ where t_i^n represents the remaining, nonsharable component of program P_i (i.e., $t_1 = t^s + t_1^n$ and $t_2 = t^s + t_2^n$).

If P_1 is shorter than P_2 ($t_1 < t_2$), then the serial schedule that minimizes average response time executes P_1 first, followed by P_2 . Under this schedule the average response time is $(t_1 + (t_1 + t_2))/2 = (3t^s + 2t_1^n + t_2^n)/2$.

If we choose to execute P_1 and P_2 jointly, then the response time is $t_{1+2} = t^s + t_1^n + t_2^n$, which is less than the average response time given by the sequential schedule iff $t^s > t_2^n$. In other words, if more than half of P_2 's execution time is sharable with P_1 , then it benefits average response time to merge P_1 and P_2 into a single jointly-executed dataflow program. In the DISC context, where file scan IO is often the dominant cost, it makes sense to merge sets of programs that read the same file.

Question 2 is more difficult to answer, because it involves reasoning about future program arrivals, e.g., using a stochastic arrival model based on the popularity of each data set. We have studied this question extensively [1]. Our main result is the following.

Suppose programs that read data file F_i arrive according to a Poisson process with rate parameter λ_i . Suppose also that the dominant cost of these programs is the IO to scan F_i , which is a large file, and hence $t_i^s \propto |F_i|$.

The quantity $|F_i| \cdot \lambda_i$ represents the *sharability* of programs that access F_i : If F_i is large, then a substantial amount of IO work can be amortized by sharing a scan of F_i among multiple programs. If programs that access F_i arrive frequently (large λ_i), then this IO amortization can be realized without imposing excessive delay on individual programs.

In our formal analysis of priority-based scheduling policies, the factor $|F| \cdot \lambda$ plays a prominent role in the priority formulae we derive mathematically, thereby confirming our intuition about sharability. Our analytical model is based on a stationary program arrival process (Poisson arrivals), but the resultant scheduling policies appear to be robust to bursty workloads. (The arrival rate parameter λ can be estimated *adaptively* based on recent arrival patterns, using standard techniques.) Our scheduling policies tend to outperform conventional ones

like FIFO and shortest-job-first, in terms of average response time, due to the ability to anticipate future sharing opportunities.

Importantly, effective sharability-aware scheduling does not depend on the ability to model the full execution time of a given program, which can be error prone. Instead, it is only necessary to model the *sharable* time t^s , which for IO-bound programs over large files is directly proportional to the file size $|F|$. This quantity can be obtained from file system metadata.

4.2 Nonconcurrent Work Sharing

We now consider ways to amortize work across programs that occur far apart in time. The mechanisms we propose can be thought of as forms of caching. Fortunately, in the DISC context most data is write-once and therefore cache consistency is not a major concern.

4.2.1 Amortizing Communication

Network IO can be amortized by caching data fragments at particular nodes in the distributed system, to avoid repeatedly transferring the same data. Let us assume that a scheduler assigns computation tasks to nodes in a manner mindful of data locality and load balancing considerations, as in the Hadoop scheduler [3]. Given such a scheduler, then ideally the placement of data fragments onto nodes should be such that the following properties hold:

1. Popular fragments, used by many programs, have replicas on many nodes. This property enables balancing of load across nodes without incurring network overhead to transfer data from busy nodes to idle ones.
2. Fragments that tend to be accessed together (many programs access the data residing in a pair of fragments, e.g., to perform a join) have some replicas co-located on the same node. This property facilitates strong locality of computation and data.
3. Popular fragments that are seldom accessed together are *not* co-located, to avoid hotspots.

The data placement problem has been studied before, with various static placement schemes being proposed and evaluated, including a model-free "round-robin" approach [20]. The round-robin tactic is consistent with the risk avoidance aspect of our philosophy (Section 1.3). However, we believe it is possible to achieve the desired data placement properties outlined above, while still avoiding reliance on explicit workload models, by making use of *adaptivity* (also mentioned in Section 1.3). Our proposed adaptive data placement scheme is:

- Each time the scheduler places a computation task on a node that does not contain all data fragments

read by the task, thereby forcing a network transfer to retrieve the needed fragment(s), store a copy of the newly-transferred fragment(s) at that node.

- Evict fragment replicas deemed to be of little utility, according to some eviction policy, subject to a constraint on the minimum replication level of each fragment for fault tolerance.

The rationale is as follows.

If the scheduler was forced to place a computation task away from its input data fragment(s), then one of three situations has likely occurred: (a) the input fragment(s) are so popular that all nodes containing replicas are busy, (b) all nodes containing replicas are busy due to the popularity of other fragments that happen to be co-located with them, or (c) no single node presently contains a copy of every fragment required by the operation (e.g., a join).

In the first case, our mechanism will increase the number of replicas of the needed fragment(s), thereby making the no-available-copies situation less likely to occur in the future. In the second case, the number of replicas will temporarily increase, followed by eviction of unused replicas on the busy nodes, resulting in a net movement of data (rather than a copy). In the third case, our mechanism brings all the required fragments to one place, so if they are accessed together again in the future full locality will be possible. If the fragments are seldom accessed independently, the eviction policy will eventually remove some of the non-co-located replicas, again yielding a net movement of data rather than a copy.

The success of this scheme hinges on the degree of temporal coherency in the pattern of programs accessing data. We are presently investigating the extent of such coherency in Yahoo!'s workloads. We are also studying the choice of eviction policy and scheduling policy, especially in regards to susceptibility to thrashing.

4.2.2 Amortizing Processing

In cases where multiple programs perform the same initial operation(s), e.g., canonicalize URL strings, remove spam pages, group pages by host, it may be beneficial to cache the result of the common operations. (The cached result may be stored on disk.) In databases, a derived data set that has been cached is called a *materialized view* [2].

One approach is to create materialized views manually, and instruct users to use these materialized views as inputs to their programs when possible. This approach is problematic. For one, selecting materialized views by hand is impractical in large, distributed organizations where no one person has a complete grasp of how data is being used around the organization. More importantly, if users start referencing materialized views explicitly in their programs, it becomes impossible to remove ones that are no longer desirable to keep. A preferred ap-

proach is for the system to select and adjust the set of materialized views *automatically*, and when applicable automatically rewrite user programs to use these views.

Standard automated methods of selecting materialized views [2] rely heavily on models to predict the size of a view, and the cost saved by using the view in lieu of the original input data. Ones that are expected to yield high cost savings relative to their size are selected.

In lieu of reliance on models, we are pursuing an approach based on adaptation and risk minimization (as motivated in Section 1.3). Our approach leverages the fact that DISC architectures store large data sets as a collection of relatively small *fragments*, each managed separately by the underlying file system. The idea is to hold small fragments of many view candidates simultaneously, continuously monitor their effectiveness, and increase the number of fragments of the most useful ones while eliminating the least useful ones. Concretely:

1. Enumerate a set of *candidate* materialized views, based on identifying common subexpressions among submitted programs. For example, if removing spam pages from an underlying web page data set is a common operation, then the spam-free derived data set is considered a candidate view.
2. Materialize a few fragments of each candidate view, e.g., materialize a spam-free copy of one fragment of the underlying web pages data set.
3. Over time, as new programs are processed by the system, compare the execution time of branches of the program that are able to leverage view fragments, to the execution time for branches that do not benefit from view fragments. The overall execution time savings attributed to a particular materialized view in a period of time is called the view's *utility*.
4. Add more fragments of views whose utility is large relative to their size, and conversely remove fragments of views with a low utility-to-size ratio.
5. Continue to adjust the set of materialized view fragments over time, as the workload evolves.

It is likely that a few of the candidate views are highly beneficial, while others are of little use or take up too much space. By materializing some fragments of each, the system ensures that it benefits at least partially from the "good" views. This approach mirrors the one advocated in investment portfolio theory for risk minimization in the presence of uncertainty. Since we do expect some degree of stationarity in our context, adaptively increasing the percentage of good-performing views in the portfolio ought to cause the portfolio to converge to a (local) optimum. Of course, the optimum may be a moving target if the workload shifts over time, which motivates continual adjustment of the portfolio.

We are currently investigating the viability of this approach, primarily from an empirical standpoint. A theoretical treatment of this problem would be of significant interest. It would likely touch upon aspects of investment portfolio theory, online learning theory (e.g., multi-armed bandit models), statistical sampling theory, and submodular optimization theory (because the utility of materializing multiple views in the same derivation chain is subadditive).

5 Summary

In this paper we discussed optimization of data-centric programs in the DISC context, a topic that has strong similarities but also important differences from database query optimization. We began by contrasting the two contexts, and offering three principles for success in the DISC context: discriminative use of information, risk avoidance, and adaptivity. Then we highlighted some database techniques that are, or can trivially be made to be, compatible with those principles.

We then turned to *cross-program optimization*. We motivated the need to optimize ensembles of interrelated dataflow programs, and argued that existing database techniques for this problem are insufficient in the DISC context. We sketched some possible approaches that to our knowledge have not been explored in either context.

Acknowledgments

We are grateful to the Yahoo! Grid team, especially the Pig engineering leads Alan Gates and Olga Natkovich, for their immense contributions to the engineering aspects of Pig. We also wish to thank Sandeep Pandey and Jayavel Shanmugasundaram, for inspiring some of the ideas in Section 4.2.2. Lastly, we thank Steve Gribble and Joe Hellerstein for helpful feedback on the paper.

References

- [1] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files, Mar. 2008. In submission.
- [2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes for SQL databases. In *Proceedings of the International Conference on Very Large Data Bases*, 2000.
- [3] Apache Software Foundation. Hadoop software. <http://lucene.apache.org/hadoop>.
- [4] Apache Software Foundation. Pig software. <http://incubator.apache.org/pig>.
- [5] R. H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. on Computing Systems*, 21(1):36–86, Feb. 2003.
- [6] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000.
- [7] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):4–24, 1990.
- [8] R. E. Bryant. Data-intensive supercomputing: The case for DISC. Technical report, Carnegie Mellon, 2007. <http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-07-128.pdf>.
- [9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, 2004.
- [10] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
- [11] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - a high performance dataflow database machine. In *Proceedings of the International Conference on Very Large Data Bases*, 1986.
- [12] P. M. Fernandez. Red brick warehouse: A read-mostly RDBMS for open SMP platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.
- [13] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1990.
- [14] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
- [15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1), 1997.
- [16] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2004.
- [17] IBM Research. Jaql software. <http://www.jaql.org>.
- [18] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.
- [19] M. Isard, M. Budi, Y. Yu, A. Birrell, and D. Fetterly. Dyad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2007.
- [20] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6(1):53–72, 1997.
- [21] Microsoft Research. DryadLINQ software. <http://research.microsoft.com/research/sv/DryadLINQ>.
- [22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.
- [23] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13(4), 2005.
- [24] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, 3rd edition. McGraw-Hill, 2003.
- [25] T. K. Sellis. Multiple query optimization. *ACM Trans. on Database Systems*, 13(1), 1988.
- [26] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the International Conference on Data Engineering*, 2003.
- [27] Yahoo! Inc. Pipes software. <http://pipes.yahoo.com>.

A TCP-layer name service for TCP ports

Sérgio Freire

PT Inovação / IEETA / Univ. of Aveiro

André Zúquete

IEETA / IT / Univ. of Aveiro

Abstract

This paper presents a simple name service for TCP ports, allowing services to be reached by name instead of number. Names are arbitrary byte arrays that are bound to listening ports. Name resolutions take place during the TCP three-way handshake, not requiring extra message exchanges. The new TCP handshake conforms with the standard and is fully compatible with existing TCP implementations. A prototype implementation was developed in Linux, paying special attention to backward compatibility with legacy systems (kernels and applications). Among the many opportunities created by the name service, it allows services with unusual names, known only by small communities, to remain undetected by port scanners (though not by network sniffers).

1 Introduction

Name systems are useful for translating user-friendly, readable strings into numerical identifiers. A popular name system is DNS [7], used for translating hierarchical, typed names into IP addresses, among other identifiers. Other name services frequently used by applications are RPC name services, such as `rpcbind` for Sun RPC and Microsoft Locator for Microsoft RPC. But up to now there is no widely used, generic name service for transport endpoints, such as TCP or UDP port names.

Historically, some TCP port names are statically bound to well known services or servers [11]. Examples are `ftp` for port 21, `telnet` for port 23, `http` for port 80, etc. This static mapping between names and ports was initially supported by local services using local data (e.g. `file /etc/services` in Unix systems). Currently there is a database service with all these static mappings [10]. However, these mappings are not mandatory; they just reflect a common use.

In this paper we propose a name service for TCP ports which enables clients and servers to resolve arbitrary names (byte arrays) to TCP ports. The advantages of

using this name service are twofold: (i) users may discriminate servers using names instead of numbers and (ii) TCP port scanners, such as `nmap`, should not be capable of discovering servers bound to unusual names.

Using names for referring ports provides a more intuitive way to refer services, instead of numbers. Service names that formerly were bound to static well-known port numbers may continue to exist but do not need any more to be bound to the same ports. For instance, we can bind the names `http` to port 8080 and `http1` to port 80. Clients access either port specifying their name, `http` or `http1`, instead of numbers 8080 and 80. Port names are also useful for uniform and uniquely tagging ports used by the servers of overlay networks.

Using arbitrary byte arrays to name TCP ports also prevents port scanning tools to discover listening ports. In fact, the success of port scanners in discovering listening ports bound to servers is due to the current small domain of port numbers — $[1, 2^{16} - 1]$. With arbitrary port names, we are able to deploy services with unusual, possibly long port names which cannot be easily found by port scanners. Services with unusual, confidential port names may be useful in many circumstances requiring restricted access profiles, namely:

- Experimental server deployment in pre-production environments;
- Private service deployment, such as personal content providers (file or web servers, mail servers);
- Restricted overlay networks.

The screening of listening TCP ports by mapping them to unusual port names is somewhat similar to the Port Knocking mechanisms [1, 4]. However, Port Knocking is an access control mechanism that requires a per-host or per-network access key. Instead, we simply require the knowledge of port names and not any key-based access control mechanisms; the knowledge of a port name is the key to access the service that uses the port.

2 Related Work

In this paper we describe a way of addressing a peer port by a name formed by a set of bytes, which acts as a weak access control mechanism – one has to know the name to access the port. Thus, in this section we briefly describe other contributions concerning these two subjects: binding of names to transport ports and access control mechanisms to transport ports.

DNS SRV records: RFC 2782 [3] defined a new type of DNS records, SRV RR, for resolving port names to a set of *(host DNS name, port number)* pairs. By creating SRV RR entries in the DNS, domain administrators are able to specify a set of locations (hosts) of a given service described by a friendly name, for the domain. For example, when the name `_foobar._tcp` is resolved in a DNS domain, it returns a set of *(host DNS name, port number)* pairs where the `foobar` service over TCP sits.

These records are useful for locating public services with well-known names in a domain, but not for dealing with arbitrary port names used in ad-hoc client-server connections, due to three reasons. First, the client application must initially learn the DNS domain of the target host before resolving the port name. For instance, to connect to port `foobar` at host `192.168.1.1`, the client must first discover the DNS domain of `192.168.1.1` (e.g. `example.org`) and then to resolve the name `_foobar._tcp.example.org`. Second, the server host must have a DNS name so that clients could match target IP addresses with host names returned by SRV RR resolutions. Finally, it requires frequent DNS updates, and the inherent administrative privileges to do so, in order to dynamically create SRV RR entries whenever servers bind names to port numbers. This is a major blocker for dynamic port name assignment and for normal end-user usage of port names.

TCPMUX: This is a service using TCP port 1 which allows a host to provide a port name handoff service for itself [5]. A client host opens a connection to port 1 on a server host and transmits the desired port name in the data stream; the server replies with a positive or negative name resolution by means of a reply character in the data stream. If the named service is available, the connection is transferred to the desired service.

In Linux systems the TCPMUX service is provided for named services handled by the `inetd` daemon. Thus, arbitrary servers cannot provide name-number bindings to TCPMUX; only servers listed in `inetd` configuration files can have named ports.

The use of TCPMUX is not transparent to clients, as they must use in a different way the connection to a server: first they must contact TCPMUX and provide the port name, then interpret the TCPMUX reply and only

afterwards, in case of success, proceed with the intended client-server interaction. Since most TCP client-server applications use a different approach, they contact directly a target server using its port number, our goal was just to improve this semantic by allowing them to use names instead of numbers to identify servers.

Port Knocking: Also known as Spread-Spectrum TCP [1, 4], Port Knocking (PK) is a mechanism for restricting access to services by allowing only authenticated requests to reach servers. It is a passive authentication scheme for TCP connections, acting as an auxiliary and external mechanism, independent of the kernel and the applications. The so called *knock* or *authentication* is typically a sequence of connection attempts to closed ports, which are intercepted and validated by a PK daemon at the destination peer. After receiving a correct *knock* from a client, the daemon allows it to connect to the wanted service port. The sequence of ports contacted in a *knock* can have an encoded meaning, like the origin IP, the remote port number and a checksum, that allows further control of the connection establishment.

Port Knocking, though a simple concept, requires: (i) a PK daemon between the client and the server; (ii) a firewall close to the server with an interaction mechanism with the PK daemon; (iii) a client application, or library functions, to carry on the *knock* before starting a connection to a protected port; and (iv) a set of closed ports for sending knock sequences.

Alternative PK implementations, using a single knock datagram (Tailgate TCP) or either an IP or TCP option (Option-Key TCP), have also been discussed in [1]. Nevertheless, the requirements are mainly the same.

Proposed Internet Draft: A discontinued IETF Internet Draft [12] proposed an extension to support TCP port names. The main goal of this proposal was to increase the number of concurrent connections for existing services by decoupling them from fixed IANA reserved port numbers. In this proposal, named server ports are in fact resolved by clients, i.e., clients proposed a resolution that is accepted by the server host if the name exists. Thus, a SYN with a server named port contains also a proposed server port name, and the returned SYN+ACK returns a name-number acceptance reply.

Port names are UTF-8 strings exchanged in a TCP header option, which strongly limits their maximum length. As a TCP header is limited to a maximum of 60 bytes, 20 of them mandatory, TCP options can only occupy 40 bytes. Moreover, part of these 40 bytes may be occupied by other TCP options, which further reduces the possible lengths of port names. This was one of the major concerns with this proposal.

Our name service is somewhat similar to this proposal but takes some different approaches. First, name to num-

ber resolutions are given by servers, and not proposed by clients, which allows us to get the same port-service decoupling on the server side if required. Second, names have arbitrary contents, they are simply byte arrays, and not just UTF-8 strings. Third, names are transmitted in the payload of TCP segments to overcome the space limitations of TCP headers. And fourth, name-number resolutions are provided to client TCP stacks, which allows future enhancements to perform semantic-aware validations (e.g. signed name resolutions, which can be validated by client applications). Therefore, our proposal is a superset of this one, in the sense that it includes all its benefits while being more flexible and powerful.

3 Proposed Name System

DNS and RPC name systems are implemented by autonomous servers, using well-known port numbers, which receive resolution requests from the application layer. Their goal is to provide a mapping from a name to a number that could be used as a parameter for lower protocol layers, namely the transport layer. Our proposal for a TCP name system goes in the opposite direction, which is to integrate it in the existing mechanism used for TCP connection establishment: the three-way handshake. Consequently, we do not use any new or existing name server, which is preferable for fault tolerance, and the name service is implemented by the TCP layer.

Besides fault tolerance, we designed the new TCP name service with the following goals in mind. The first one was to maintain compatibility with existing TCP/IP stacks. The second one was to add a new name service, and not to replace the current addressing mechanism used by TCP segments, with port numbers, by names, as numbers are more efficient to handle than names. The third one was to allow TCP clients and servers to use arbitrary name formats, in order not to restrict future uses by upper protocol layers. We also foreseen another goal, which is not covered in this paper, which is to add arbitrary semantics to name resolution (e.g. versions). In this paper we only have one simple, default semantic: strict byte equality between names.

3.1 Name binding

Our TCP name service allows TCP servers to bind names to listening ports and clients to use port names when requesting a TCP connection. The port name is associated with the service/application during the socket binding procedure at the server side. Clients refer the port name when they specify the TCP address of the server.

As we just want to provide a TCP name service, and not to fully replace port numbers by port names, port names must always be associated to port numbers. Therefore, the modified TCP layer on the server side will keep a port number to each local port name. When a

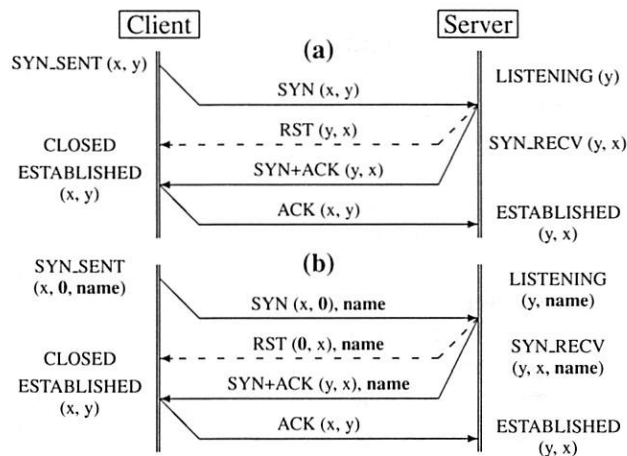


Figure 1: Standard 3-way handshake, using port numbers x and y (a) and extended 3-way handshake using a server port name (b). The slashed line represents an alternative server reply (RST segment) when the connection to port y or with the given name is rejected.

application binds a name to a TCP endpoint (socket), it immediately gets a number as well. For backward compatibility, applications may specify the port number; if not specified, the TCP layer allocates a free port number.

For applications binding only names to ports, and not fixed numbers, the TCP layer can allocate random port numbers on a per-request basis. The benefits of this protocol decoupling from fixed port numbers are (i) harder traffic eavesdropping and (ii) increased number of concurrent connections, as in [12].

Port names are resolved as soon as possible to allow clients and servers to use port numbers in the normal TCP stream exchanges. Consequently, we integrated the name resolution in the TCP synchronization phase, where clients and servers exchange initial sequence numbers and TCP options (see Fig. 1). The port name is specified in the SYN request, the name→number resolution is given in the subsequent SYN+ACK segment. Thereafter, both peers will always use the server port number in all segments exchanged in the TCP stream. Stateful firewalls should have no problems managing this, if properly updated to keep track of port number/name pairs.

The name resolution works as follows. The server host, when receiving a SYN with a port name, ignores the destination port number and looks for a socket in the LISTEN state bound to that port name. If such a socket is found, a SYN+ACK is sent to the client containing the name resolution, i.e. port name and number. Otherwise, a RST packet is sent with the unresolved port name.

3.2 Backward compatibility

Port name resolutions, expressed in SYN requests, should carry a null server port number to force a RST reply from

standard TCP stacks. Though the TCP standard does not refer that 0 is an invalid port number, in practice it is not used; therefore, it can be used for differentiating old TCP stacks from new ones implementing port names. Protocol scrubbers [6] should be updated to include TCP destination port 0 as valid when port name option is present.

Port name resolutions are only provided to clients that request a connection to a TCP endpoint with a port name. Clients using the normal TCP connection request, i.e. using a server port number, do not get any name resolution when the port number is actually associated with a name. Likewise, they do not receive a port name in a RST reply. This is done for two reasons. First, the client didn't request a name resolution, so it should not get one. Second, for backward compatibility with current TCP stacks, clients using the actual number-based port addressing should observe the standard TCP behavior from servers (described in Fig. 1).

3.3 Port names in TCP segments

As above described, port names are only used in the TCP synchronization phase, therefore in SYN, SYN+ACK and RST segments. Thus, we have to conceive a way of adding arbitrary names to these segments and to signal that they should be used.

As referred in §2, adding a name to a TCP header is not a suitable solution, because it severely limits the length of port names. Instead, we decided to use the TCP payload to store the port name and to use a new TCP option to signal the presence and the length of the port name in the beginning of the segment payload.

TCP synchronization segments allow clients and servers to exchange data. Though this is not supported by the Berkeley sockets API, it is allowed by the XTI (X/Open Transport Interface) and is a correct behavior according to the standard [9]. RST segments usually do not carry any data in their payload but a standard amendment [2, §4.2.2.12] defines that they may carry a reason message in their payload. Thus, using the payload of synchronization and reset segments for exchanging port names is correct, according to the standards, though such data should not be delivered to upper layers.

As port names are transmitted in the segments' payload, they have a direct effect in the management of the stream sequence numbers. The sequence and acknowledge numbers exist to control the data stream between peers, guaranteeing byte order and resilience to data loss.

We decided to keep the semantics of sequence numbers during the synchronization using port names, i.e. names are seen as data exchanged in payloads (see table below), but they are removed from the data that is provided to upper protocol layers. This does not raise any coherence problem, since upper layers are not aware of sequence numbers. In other words, upper layers do

not notice that the amount of data received in segments' payload is not the same amount of data they actually get.

Consequently, whenever a name resolution is required in a SYN segment, its reply, SYN+ACK or RST, acknowledges a sequence number equal to the client's ISN plus the port name length plus one. Similarly, the client's ACK will contain a sequence number equal to its ISN plus the server's port name plus one:

Segment	TCP standard		TCP with port names	
	seq	ack	seq	ack
SYN	ISN _c	0	ISN _c	0
RST	0	ISN _c + 1	0	ISN _c + 1 + L
SYN+ACK	ISN _s	ISN _c + 1	ISN _s	ISN _c + 1 + L
ACK	ISN _c + 1	ISN _s + 1	ISN _c + 1 + L	ISN _s + 1

where L is the length of the port name.

3.4 Caching of name resolutions

Some name services' clients maintain caches of name resolutions, which is common in DNS but not in RPC. We decided not to maintain caches in our TCP port name resolutions. Two main reasons justify our decision. First, name resolutions are piggybacked in the first two segments of a TCP synchronization, thus the overhead of name resolution is too reduced to justify the existence and management of caches in clients for increasing performance. Second, stalled cached resolutions can lead to wrong TCP connections that only applications can possibly detect, but not the TCP layer.

3.5 Managing port access restrictions

The TCP name service prevents services with unusual names to be discovered by port scanning tools, hiding them from people or tools that wish to exploit and not really use them. But in §3.1 we saw that port names may be associated to fixed port numbers. Thus, to enforce the name-based access control we need to disallow clients to connect server ports using only their number. So, servers must specify, when binding, if (i) port-based connections are permitted or (ii) only name-based connections are allowed. The latter allows the TCP layer to use random numbers for the server port on each connection request.

3.6 Denial of Service (DoS) issues

Name resolutions in servers require more time than simple port number lookups. However, the extra workload required to resolve port names does not prevent other clients from accessing the server or even local server applications to run. Thus, though name resolution flooding attacks may slow down servers, by itself they do not create a clear and well defined DoS scenario.

4 Implementation

Our TCP name service was implemented in a recent Linux kernel (2.6.22.9), using a Fedora 7 kernel source.

In Linux there are two separate implementations of TCP, one for IPv4 and another for IPv6. Since our implementation is mainly a proof of concept, we only updated the TCP version for IPv4. Nevertheless, we took into consideration some IPv6 issues, as for the naming of TCP endpoints, described in §4.7.

4.1 Socket related structures

Port names, either bound to local ports or to be resolved within connection handshakes, are stored in dynamically allocated, kernel space memory areas. Structures storing port numbers, `inet_sock` and `tcp_sock`, were updated to include an optional port name; the latter was also updated to include port access restrictions.

Some auxiliary structures, `tcp_request_sock` and `tcp_options_received`, were also enriched to maintain the length of the port name. This value simplifies the calculation of sequence numbers in TCP segments containing port names and helps locating the socket structure in hashed lists (see §4.3).

4.2 Name→number mappings

Since port names are just a step towards port numbers, some mapping table converting names to numbers must exist. This mapping only applies for local ports, as no caching of remote name-number bindings is required. We implemented this mapping by extending the `inet_hashinfo` structure to include an hashed variable based on a new structure named `portname_hashbucket`:

```
struct portname_hashbucket {
    spinlock_t      lock;
    struct list_head list;
    unsigned short   port;
    char             * portname;
    unsigned short int portname_len;
};
```

An element is first added to this hashed variable when a socket is bound to a port name, in `inet_bind`. This is implemented by a new function, `inet_bind_hash_portname`, which extends the functionality of `inet_bind_hash`. An element is removed from the hashed variable when the socket referring it is eliminated in `inet_unhash`.

This variable is used for name lookup in two distinct occasions: (i) when binding a name to a port, in `inet_csk_get_port`, to check if it is not already being used, and (ii) in `inet_lookup_listener`, to get the port number of a listening socket upon receiving a SYN with a port name.

4.3 Socket hashed lists

Linux implements several hashed lists to index sockets. One of them is `listening_hash`, containing `INET_LHTABLE_SIZE` lists of sockets involved in TCP connections. The actual list of a socket is given by the functions `inet_ehashfn` and `inet_sk_ehashfn`, which produce

an integer from the source/destination addresses and port numbers. Thus, both local and remote port numbers are crucial for indexing sockets in `listening_hash`.

However, clients using name-based connections raise a problem: they don't have a remote port number when adding a socket in the `SYN_SENT` state, i.e. after sending a SYN segment with a port name. Thus, for these client sockets there is a temporary hashing within `listening_hash` until getting the name resolution. This temporary hashing applies only to sockets in the `SYN_SENT` state; after getting a correct SYN+ACK segment with the required name resolution, the remote port is used to relocate the socket, now in the `ESTABLISHED` state.

For the temporary hashing we used the same functions and replaced the server port number by the port name length. We could as well have used a null server port number but using the name length is more likely to improve, with no extra costs, the spreading of sockets (only in the `SYN_SENT` state) among the hashed lists when many port names are used.

4.4 Defining port access restrictions

Implementing TCP port access restrictions is accomplished by setting a new, TCP level socket option. We named this option `TCP_BIND_PORTNAME` and gave it the value 15. There are three different listening modes that can be set through this option: (0) port number only (current standard); (1) port name only (legacy connection requests are not allowed); and (2) port number and port name: both legacy and name-based SYN requests are accepted. This option is checked by `tcp_v4_rcv` function, upon the arrival of a SYN segment.

4.5 Port names in TCP segments

When a client application issues a connection request using a port name, the local TCP stack copies the port name from the `sin_portname` field of the provided `sockaddr_in_named` structure and updates internal variables as needed. The kernel will then send a SYN packet with the port name in the payload and a TCP option indicating the length of the port name, which corresponds to the given `sin_portname_len`. We used the value 0x45 for the new TCP option, which uses a 16-bit integer to communicate the port name length.

As the Linux TCP does not handle user data in synchronization segments, no modifications were required to prevent port names exchanged in SYN and SYN+ACK segments to be delivered as normal data to applications.

Linux has one special socket `tcp_socket` that is only used for sending RST segments by the function `tcp_v4_send_reset`. This function and `ip_send_reply`, called to generate the actual RST IP datagram, were extended to process further parameters, namely port names, and to handle port names in TCP segments.

4.6 IP fragmentation

Linux sets the Don't Fragment IP flag in TCP segments not containing payload, such as SYN and SYN+ACK, which is a correct behavior [8] since their packet length is below the fragmentation threshold of 68 bytes. But with our port name resolution their payload contains a (possibly large) port name, thus fragmentation must be allowed in those cases. The function *tcp_transmit_skb* was modified, when calling *ip_queue_xmit*, to allow IP fragmentation for this synchronization segments.

4.7 Using TCP port names

For binding names to TCP ports and to express port names when connecting to them, we created two new structures by extending *sockaddr_in* and *sockaddr_in6* structures, for IPv4 and IPv6, respectively. The new types were created by adding two extra fields: a pointer to the port name and the port name length, see below.

```
struct sockaddr_in_named {
    sa_family_t    sin_family;      /* AF_INET_NAMED */
    in_port_t      sin_port;        /* port number */
    struct in_addr  sin_addr;        /* IP address */
    unsigned char  sin_zero[2];
    uint16_t       sin_portname_len; /* Port name len */
    char __user    *sin_portname;   /* Port name */
};
```

Furthermore, we created two new address families for using with these new structures: *AF_INET_NAMED* for IPv4 and *AF_INET6_NAMED* for IPv6. At kernel level, these new address families are used solely for identifying the type of naming structures provided by client applications; for all other family tagging requirements, it uses the families *AF_INET* and *AF_INET6*.

The function *inet_bind* was extended to support binding to a local port name. Similarly, the functions *tcp_connect* and *tcp_v4_connect* were extended to support the connection to named ports.

5 Experience

For evaluating the features described in this proposal, we implemented some basic TCP client/server programs using the new *sockaddr_in_named* structure and ran them in a server with our modified Linux kernel. We also patched some well known applications, such as Apache2 web server and netcat. Apache2 was partially patched to bind to a specific port name; netcat was used as a basis to build a command line application supporting both port number and port name bindings.

All possible combinations of server port name bind modes with client connection methods were tested successfully within and between machines with the standard and enhanced TCP stacks, as shown in the table below. Clients with old TCP stacks can only connect to

port numbers (cases 3 and 4) and servers with old TCP stacks can only handle connection requests including a valid port number (cases 2 and 4). Clients with new TCP stack (cases 1 and 2) can either connect by port number or name but the latest will only be fully understood by the new TCP port name aware stack (case 1).

		Server-side stack	
		new	old
Client-side stack	new	✓ ¹	✓ ²
	old	✓ ³	✓ ⁴

6 Conclusions

In this paper we presented a simple name service for TCP ports. This name service allows TCP clients to identify services by port name, instead of port number, which is more user-friendly. The name service extends TCP synchronization segments, conforms with the TCP standard and is compatible with existing TCP implementations. Resolution of port names requires additional processing but does not create an opportunity for DoS attacks.

A security advantage of TCP port naming is that it allows services with unusual names, known only by small communities, to remain undetected by port scanners.

Our prototype implementation confirmed the compatibility with other TCP implementations. Furthermore, we were able to maintain compatibility with legacy systems, kernels and applications.

References

- [1] BARHAM, P., HAND, S., ISAACS, R., JARDETZKY, P., MORTIER, R., AND ROSCOE, T. Techniques for Lightweight Concealment and Authentication in IP Networks. Tech. Rep. IRB-TR-02-009, Intel Research Berkeley, 2002.
- [2] BRADEN, R. Requirements for Internet Hosts – Communication Layers. RFC 1122, IETF, Oct. 1989.
- [3] GULBRANDSEN, A., VIXIE, P., AND ESIBOV, L. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, IETF, Feb. 2000.
- [4] KRZYWINSKI, M. Port Knocking: Network Authentication Across Closed Ports. *SysAdmin Magazine*, 12 (2003), 12–17.
- [5] LOTTOR, M. TCP port service Multiplexer (TCPMUX). RFC 1078, IETF, Nov. 1988.
- [6] MALAN, G. R., WATSON, D., JAHANIAN, F., AND HOWELL, P. Transport and application protocol scrubbing. In *INFOCOM (3)* (2000), pp. 1381–1390.
- [7] MOCKAPETRIS, P. Domain names – implementation and specification. RFC 1035, IETF, Nov. 1987.
- [8] POSTEL, J. Internet Protocol. RFC 791, IETF, Sept. 1981.
- [9] POSTEL, J. Transmission Control Protocol. RFC 793, IETF, Sept. 1981.
- [10] REYNOLDS, J. Assigned Numbers: RFC 1700 is Replaced by an On-line Database. RFC 3232, IETF, Jan. 2002.
- [11] REYNOLDS, J., AND POSTEL, J. Assigned Numbers. RFC 1700, IETF, Oct. 1994.
- [12] TOUCH, J. A TCP Option for Port Names. Internet draft (expired), IETF, Apr. 2006.

Using causality to diagnose configuration bugs

Mona Attariyan and Jason Flinn
Computer Science and Engineering
University of Michigan

Abstract

We present a novel method for diagnosing configuration management errors. Our proposed approach deduces the state of a buggy computer by running predicates that test system correctness and comparing the resulting execution to that generated by running the same predicates on a reference computer. Our approach generates signatures that represent the execution path of a predicate by recording the causal dependencies of its execution. Our results show that comparisons based on dependency sets significantly outperform comparisons based on predicate success or failure, uniquely identifying the correct bug 86–100% of the time. In the remaining cases, the dependency set method identifies the correct bug as one of two equally likely bugs.

1 Introduction

Software in modern computer systems is extraordinarily complex. Many applications have a large number of configuration options that can customize their behavior. Further, each application interacts with the other software on a computer through channels such as shared libraries, registry entries, environment variables and shared configuration files. This flexibility has a cost: when something goes wrong, fixing a configuration problem can be both time-consuming and frustrating. Consequently, there has been considerable effort by the research community to simplify configuration management [2, 5, 8, 9, 10, 11].

The process of configuration management can be divided into two separate tasks: diagnosing which specific problem is afflicting the computer system, and determining how to fix that problem. In this paper, we address the former task: finding the root cause of a configuration problem. We assume that the bug is known, i.e., the problem has been previously encountered and solved on a *reference computer*. The reference computer could be a test system used by software developers or a personal computer owned by a peer who had the same problem. Thus, the problem of diagnosing an unknown bug on a *sick computer* can be reduced to identifying that the sick computer is in a state similar to a buggy state on the reference computer for which a solution is known.

To deduce similarity between states on the reference and sick computers, our approach is to run a set of *predicates* that test the correctness of the computer system. In previous work [8], we used the success or failure of predicates to deduce similarity. While this approach is intuitive, we have since encountered several drawbacks. First, an expert, e.g., a software developer or tester, must craft a predicate to cover each new bug. Second, a single predicate may often detect many bugs, causing many states to appear similar. Finally, a test case that is too finely crafted to the reference computer may inadvertently report an error due to a benign difference between the environments of the sick and reference computers.

We present a method for diagnosing bugs that uses signatures derived from the set of objects upon which each predicate's execution causally depends. We use system call tracing tools such as `strace` to record each predicate's *dependency set*, i.e., the files, devices, fifos, etc. read by the predicate. We compare the dependency sets generated on the reference and sick computers to deduce similarity. Our results show that comparisons based on dependency sets significantly outperform comparisons based on predicate success or failure, uniquely identifying the correct bug 86–100% of the time. In the remaining cases, the dependency set method identifies the correct bug as one of two equally likely bugs.

2 Background

Our previous work in configuration management, titled AutoBash [8], used the pattern of success and failure of known predicates to diagnose configuration errors. Using this approach, AutoBash executes all predicates, $\{P_0, P_1, \dots, P_n\}$ on the sick machine and aggregates their results as a binary vector $S_{current} = \{1, 0, \dots, 1\}$ (with 1 indicating success and 0 failure). AutoBash then compares $S_{current}$ with a set of system state vectors S_i from $\{S_0, S_1, \dots, S_m\}$, where each system state was generated by running the predicates on the reference computer prior to fixing a known bug. Intuitively, each vector is a signature for a system state that represents a particular bug. Thus, AutoBash chooses the system state vector that is most similar to $S_{current}$ as the most likely diagnosis for the bug. According to the diagnosis, AutoBash chooses a solution from its database and speculatively runs the

solution. Then, AutoBash tests the affected predicates to determine whether the problem is fixed or not. If the problem is fixed, AutoBash commits the solution; otherwise, the solution is rolled back and AutoBash tries the next most likely diagnosis. The accuracy of diagnosis determines how fast AutoBash can find a correct solution. As the AutoBash diagnosis method uses the Hamming distance as a similarity metric, we will refer to it as the *Hamming distance method*.

One advantage of the Hamming distance method is that it treats predicates as black boxes. AutoBash does not need to understand what each predicate does; it only needs to execute each predicate as a child process and check the return code to determine success or failure. Another advantage is portability; since predicates are application-level test cases, their success or failure should not be perturbed by irrelevant fluctuations in the application environment such as variations in the operating system or installed software.

However, as Section 5 shows, the Hamming distance method suffers from ambiguity. Since the similarity metric takes into account only the success or failure of predicates, many different bugs may have identical state vectors. To allow correct diagnoses, a tester or developer must painstakingly craft specific predicates that target each known bug. Easy-to-create stress tests, which we refer to as *kitchen sink predicates*, are useless because they fail for most bugs. For example, a Linux kernel compile can trigger many possible compiler configuration bugs, so its failure tells little about the underlying system state. On the other hand, failure of a hand-crafted predicate that only checks a specific kernel header reveals much more about the bug. However, writing such predicates to cover all known bugs takes a lot of effort.

Another drawback of the Hamming distance method is lack of granularity: many system state vectors may lie at a Hamming distance of one or two from a given result vector, even though each state causes a different set of predicates to fail.

3 Design

Based on our observations, we tried to design a method that would retain the advantages of the Hamming distance method while eliminating its disadvantages.

Looking more closely, we realized that although the success or failure of predicates may be similar for many bugs, the execution paths of those predicates usually differ for each bug. For example, if a predicate compiles and runs a program, any bug in the compilation, linking or loading phases can cause the predicate to fail. However, bugs in each of the three phases cause the predicate to take different execution paths. As another example, a configure script takes different execution paths depending upon the particular software that is installed on a computer. Thus, if we can generate a signature that captures the execution path of a predicate, we should be able to more precisely identify a configuration error.

Ideally, we would like to generate a signature that is precise enough to capture different execution paths that are induced by different configuration bugs. However, the signature should be robust enough so that executing a predicate on computers with the same bug but different operating systems, installed software, and execution environments generates similar signatures. For example, we could use all the system calls executed by a program to generate a signature for the execution path [4, 12]. However, random permutations caused by thread scheduling, interactions with other processes, and other sources of non-determinism will cause the sequence of system calls to vary even when a predicate is executed on the same platform. Further, this method would perform poorly for our purposes because we run the same predicate on two computers with different software. For example, the sequence of system calls will change with different versions of shared libraries such as *libc*, with different versions of the same operating system, or with different operating systems.

To generate a more robust signature, we decided to instead use the causal dependencies of predicate execution as a signature. We define the dependency set of a process to be the set of files, directory entries, file metadata, devices, fifos, and other objects read by the process and its descendants during their execution. This choice is based on the observation that the layout of application files and directories shows only minor fluctuations across platforms. Further, the concept of files and directories is common to most operating systems, while specific system calls differ greatly. At the same time, the dependency set usually reflects significant differences in the execution paths of a predicate in the presence of different bugs. For instance, in the above compilation example, if the predicate fails in compilation, the predicate's dependency set will not contain any objects related to the linker or loader simply because execution ended before those phases. Therefore, the dependency set can capture the progress of predicate execution and generate different signatures for different failures.

There are several possible approaches for generating dependency sets. We wished to avoid intrusive monitoring methods that require the application under test or the host operating system to be modified. We also wanted to reuse existing tools as much as possible. We observed that most operating systems have a system call tracing tool such as Linux's *strace* or FreeBSD's *ktrace*. We wrote parsing programs that take tracing tool output and generate the corresponding dependency set. The only drawback of these tools is that they can only trace the main process and its descendants. Activities of other processes communicating with the main process and its descendants via shared memory, pipes or files cannot be automatically traced with these tools. To address this issue, we could trace all processes in the system. However, we judged that tracing all processes would incur a lot of overhead while adding negligible accuracy.

4 Implementation

We use `strace` and `ktrace` to generate dependency sets on Linux and FreeBSD, respectively. These tools intercept all system calls made by a process and its descendants along with their parameters and return values. We trace each predicate and pipe the tool output to a parser that calculates the predicate's dependency set.

The parser divides system calls into three categories. The first category consists of system calls that do not affect the dependency set of the predicate. For example, the `brk`, `mmap` and `mprotect` system calls manage a process's memory. The parser simply ignores these system calls. The second category consists of system calls that do not directly affect the dependency set but may change the objects that are added later. For example, the `fchdir` system call changes the current directory to the file descriptor specified by its first parameter. This system call does not change the dependency set, but it affects all following file names with relative paths.

The third category consists of system calls that directly affect the dependency set. For each system call, the parser adds appropriate dependency records to the process's dependency set. For example, the `stat` system call provides information about a specified file. A successful `stat` system call makes the process dependent on the directory entry and metadata of the specified file, as well as the directory entries and metadata of all directories in the file path. As another example, reading from a file makes a process dependent on the content of the specified file, as well as its metadata.

Before processing the parameters of a system call, we check the return value and error type. Without considering the return value, we are in danger of adding wrong records to the dependency set. For example, `ENOENT` as the return value of an access system call indicates that the requested path does not exist or is a dangling symbolic link. Therefore, we cannot simply generate dependency records for the entire path. Instead, we determine which part of the path exists and add appropriate dependency records for only that part.

Usually, the main process creates child processes using `fork`. Our parser tracks dependency sets for the descendants of a traced process in order to generate a good signature. For example, a `make` process forks children to compile and link objects; if these child processes were omitted, the resulting dependency set would contain little useful information.

Initially, the parser sets the dependency set of a child process equal to the dependency set of its parent. It adds new records to the child's dependency set as the child executes. If the child communicates to its parent (e.g., by sending the parent a signal when it exits), the parser sets the dependency set of the parent process to be the union of the parent's current dependency set and the child's dependency set. The `fork` system call is usually followed by an `exec` system call that replaces the memory image of the process with one from an executable

file. When this happens, the parser adds the executable file to the process's dependency set.

In our current implementation, the parser uses full path information for files and directories. We also considered using only the name of a file or directory instead of the whole path. However, our experiments revealed that the former method was slightly superior, mainly due to false matches between files with the same name but different paths. We did find that using only the file name was especially useful for shared libraries, because the location of libraries can vary widely across platforms. Therefore, our implementation uses only the file name for shared libraries. Our parser has one further optimizations: if an object being read is referred to by a symlink, the parser follows the symlink to also add entries for the real path of the object.

To diagnose a configuration error on a sick computer, our tool runs each predicate, traces its output, and generates its dependency set. It compares the dependency sets with those generated on the reference computer for each known bug. To compare dependency sets, the tool calculates the edit distance between the sets for each predicate. For each known bug, it sums the edit distances to calculate the similarity between the state of the sick computer and the state of the reference computer. It identifies the bug with the lowest total as the most likely diagnosis; in the case of ties, it reports all tied bugs as being equally likely to be the root cause.

5 Evaluation

Our evaluation measures how effectively our proposed dependency set method diagnoses configuration bugs using both targeted and "kitchen sink" predicates.

5.1 Methodology

In previous work [8], we developed a benchmark consisting of three applications: the CVS version control system, the gcc cross compiler and the Apache Web server. For each application, the benchmark consists of 10 common configuration bugs. It also contains 5–6 targeted predicates for each application such that each bug causes at least one predicate to fail. Although a complete description of our benchmark is omitted due to space constraints, Table 1 shows some examples of bugs and predicates. In addition, for each application we created a single "kitchen sink" predicate that detects all bugs.

In order to measure how sensitive our dependency set method is to variation across operating systems and installed software, we ran our experiments on four computers running different operating systems: Red Hat Enterprise Linux 3, Fedora core release 6, Ubuntu version 7.04, and FreeBSD version 6.2. Although these platforms are fairly similar in overall behavior, the execution signatures revealed a lot of subtle differences. For instance, in our Ubuntu platform libraries are located in `/lib/tls/i686`, while in other systems `/lib` contains the libraries. As another example, FreeBSD

Application	Configuration problem descriptions
CVS	Setgid bit not set on repository, so group for new files is incorrect
	\$CVSROOT misconfigured for a CVS user
GCC	Cross-compiler not configured for -pthread flag
	Cross-compiler not configured to pass the static link flag to the linker
Apache	Apache configuration does not allow CGI execution in user's home directory
	Apache not configured to load PHP module
Application	Predicate descriptions
CVS	a user checks in a project and checks it out again
	a user checks in a project, and a different user checks it out
GCC	compile a .c file and statically link in a math library
	take a multi-threaded .c file, compile it for the XScale architecture
Apache	wget a CGI script from a user's home directory
	wget the result of a PHP test page

Table 1. Example bugs and predicates

uses “/etc/pwd.db” and “/etc/spwd.db” for authentication, while other platforms use “/etc/passwd”. We installed the same version of CVS and the gcc cross compiler on all machines. For Apache, we used version 2.0.50 for all machines, except for FreeBSD, which runs 2.0.59. The version of the PHP module that we used is 4.4.6, except for Fedora, which runs 4.4.7.

We used the Red Hat machine as the reference computer. For each application, we injected each bug. We then executed the targeted predicates and recorded the success or failure of each one, as well as its dependency set. We also executed the “kitchen sink” predicate for each bug, recording its outcome and dependency set.

We emulated sick computers by injecting each bug into all four computers. For each bug, we ran the targeted and “kitchen sink” predicates on each sick computer and used both the Hamming distance and dependency set methods to diagnose the bug. Each method returns a set of bugs that are judged to be the root cause of the configuration problem. Multiple bugs are returned by each method only in the case of ties, where each bug is judged equally likely to be the root cause. Two bugs of the benchmark (CVS bug 4 and Apache bug 4) were not applicable to FreeBSD platform due to differences in platform default behavior and application versions, so we omitted these bugs from our results.

We evaluated our results using two metrics from the information retrieval literature: precision and recall. Precision, which is the percent of false positives, is calculated as $|R \cap C|/|R|$, where R is the set of bugs returned by a method and C is the set of bugs that are the correct root cause. Recall, which is the percent of false negatives, is calculated as $|R \cap C|/|C|$.

5.2 Results

Table 2 shows results for the targeted predicates. We only show precision in the table since both the Hamming distance and dependency set methods have a recall of 100%, i.e., there were no false negatives in our experiments. Because the Hamming distance method only

considers the success or failure of predicates, its results are the same on all sick computers. Therefore, we only show its precision once in the third column of the table. The remaining columns show the precision of the dependency set method on each sick computer.

As the third column of Table 2 shows, the Hamming distance method performs fairly well as long as an expert has taken the time to write targeted test cases. However, this method only considers the success or failure of predicate execution. Therefore, it cannot distinguish between situations with identical fail/pass patterns. Although our benchmark consists of targeted predicates, the Hamming distance algorithm still generates many ties. Across all bugs, its average precision is 57%.

As the remaining columns in the table show, the dependency set method has greater precision. On the Red Hat platform, the sick computer is identical to the reference computer. Thus, the dependency set method acts like an oracle, having precision of 100% for all bugs. For the remaining platforms, the dependency set method has average precision of 93%.

Table 3 shows results for the “kitchen sink” predicates. As before, neither method generates false negatives. However, the Hamming distance method has low precision for all bugs. It does not provide any useful information because kitchen sink predicates always fail. In contrast, the dependency set method is able to diagnose bugs much more accurately. The average precision of the dependency set method ranges from 93% to 100%, compared to 10% for the Hamming distance method. These results show that the dependency set method can still do an excellent job of diagnosing bugs without requiring the time-consuming task of writing targeted predicates.

The overhead of generating dependency sets is very small. On average, it takes less than 0.2 seconds to generate a signature from each trace output. Overall, it takes less than 14 seconds for CVS, 11 seconds for gcc and 27 seconds for Apache to run all the predicates under trace and generate a complete signature. In our experiments, the time required to compare the complete signature of a sick computer against the reference computer is less than 0.5 seconds. As the number of predicates and bugs in the database increases, the time required for generating the complete signature and comparing against the reference machine increases as well.

The accuracy of our method is dependent on the distance between bugs rather than the size of bug database. In other words, our method cannot accurately distinguish between bugs that are subtly different from each other and cause predicates to have similar executions. Although the chance of having such bugs increases as the database grows, the size of the database does not solely determine the precision of our method.

6 Related work

To the best of our knowledge, this work is the first to use the causal dependencies of predicate execution to

Application	Bug	Hamming distance	Dependency set (RHEL 3)	Dependency set (Fedora)	Dependency set (Ubuntu)	Dependency set (FreeBSD)
CVS	1	100%	100%	100%	100%	100%
	2	33%	100%	50%	50%	50%
	3	100%	100%	100%	100%	100%
	4	33%	100%	100%	100%	N/A
	5	100%	100%	100%	100%	100%
	6	33%	100%	100%	100%	100%
	7	33%	100%	50%	50%	50%
	8	33%	100%	50%	50%	50%
	9	100%	100%	100%	100%	100%
	10	33%	100%	50%	50%	50%
gcc	1	50%	100%	100%	100%	100%
	2	50%	100%	100%	100%	100%
	3	100%	100%	100%	100%	100%
	4	33%	100%	100%	100%	100%
	5	33%	100%	100%	100%	100%
	6	100%	100%	100%	100%	100%
	7	50%	100%	100%	100%	100%
	8	50%	100%	100%	100%	100%
	9	100%	100%	100%	100%	100%
	10	33%	100%	100%	100%	100%
Apache	1	100%	100%	100%	100%	100%
	2	100%	100%	100%	100%	100%
	3	20%	100%	100%	100%	100%
	4	20%	100%	100%	100%	N/A
	5	20%	100%	100%	100%	100%
	6	50%	100%	100%	100%	100%
	7	50%	100%	100%	100%	100%
	8	100%	100%	100%	100%	100%
	9	20%	100%	100%	100%	100%
	10	20%	100%	100%	100%	100%

Table 2. Precision of bug diagnoses for targeted predicates

Application	Bug	Hamming distance	Dependency set (RHEL 3)	Dependency set (Fedora)	Dependency set (Ubuntu)	Dependency set (FreeBSD)
CVS	1	10%	100%	100%	100%	100%
	2	10%	100%	100%	100%	50%
	3	10%	100%	100%	100%	100%
	4	10%	100%	100%	100%	N/A
	5	10%	100%	100%	100%	100%
	6	10%	100%	100%	100%	100%
	7	10%	100%	50%	50%	50%
	8	10%	100%	50%	50%	50%
	9	10%	100%	100%	100%	100%
	10	10%	100%	50%	50%	50%
gcc	1	10%	100%	100%	100%	100%
	2	10%	100%	100%	100%	100%
	3	10%	100%	100%	100%	100%
	4	10%	100%	100%	100%	100%
	5	10%	100%	100%	100%	100%
	6	10%	100%	100%	100%	100%
	7	10%	100%	100%	100%	100%
	8	10%	100%	100%	100%	100%
	9	10%	100%	100%	100%	100%
	10	10%	100%	100%	100%	100%
Apache	1	10%	100%	100%	100%	100%
	2	10%	100%	100%	100%	100%
	3	10%	100%	100%	100%	100%
	4	10%	100%	100%	100%	N/A
	5	10%	100%	100%	100%	100%
	6	10%	100%	100%	100%	100%
	7	10%	100%	100%	100%	100%
	8	10%	100%	100%	100%	100%
	9	10%	100%	100%	100%	100%
	10	10%	100%	100%	100%	100%

Table 3. Precision of bug diagnoses for kitchen sink predicates

diagnose configuration bugs. Previous systems have used predicates to help fix buggy computers. Chronus [11] also uses user-defined predicates to test the behavior of

the system. Chronus tries to find the point in time where a system ceased to operate correctly by testing a predicate against different virtual machine snapshots. The

success or failure of the predicate is assumed to precisely diagnose the bug. We must avoid this assumption in order to eliminate having an expert write a targeted predicate for each new bug. Since Chronus compares the system against itself, it is able to diagnose unknown bugs. Our method, however, cannot diagnose bugs that do not exist in the reference computer database. Our previous work, AutoBash [8], also used predicates but employed the Hamming distance method discussed in the paper.

PeerPressure [9] and its predecessor, Strider [10], also address the configuration management problem. These tools apply statistical methods to diagnose and fix configuration problems. PeerPressure and Strider benefit from the known schema of the registry, but cannot detect configuration errors that lie outside the registry. Our approach of analyzing predicate causality is more general and holds promise for dealing with errors that lie outside the registry and on other operating systems such as Unix variants. We assume that the bug is already known and exists in the reference computer database, but PeerPressure and Strider do not have this assumption.

Clarify [3] uses a similar approach of generating signatures that are based on program behavior. Clarify targets improved error reporting rather than configuration management. Clarify generates signatures using program features such as function call counts, call sites, and stack dumps. It then classifies the signatures using machine learning techniques. In contrast, our approach uses causal dependency information from a tree of processes to generate a signature.

Similar to our method, Yuan *et al.* [12] leverage system call information to diagnose configuration bugs. They correlate system call traces to problem root causes using machine learning techniques. To reduce system call variations, they use cross-time and cross-machine noise filtering techniques. Our method generates more robust signatures by extracting dependency sets from system call traces. The dependency set method does not need cross-time filtration and is accurate across variations of Unix operating systems.

Many prior systems use causality analysis. For instance, BackTracker [6] traces causality to determine what state has been changed during an intrusion. Aguilera *et al.* [1] trace RPCs to debug performance problems. PASS [7] uses causality to annotate files with provenance that describes their causal inputs. Our dependency sets capture similar information, but limit the scope of information collected to specific periods of time.

7 Conclusion

Configuration management is a difficult problem that is taking on increased importance as the complexity of modern computer systems grows. This paper contributes a novel method for error diagnosis that uses the causal dependencies of test case execution to detect similarities between a configuration state on a sick computer and another on a reference computer. We show that such

information can be collected using only pre-existing system call tracing tools and without requiring application or operating system modification. As future work, we would like to measure the robustness of our signatures across different versions of the same application.

Acknowledgments

We thank Ya-Yunn Su and Kaushik Veeraraghavan for comments on this paper. This work has been supported by the National Science Foundation under award CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-0346686. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

References

- [1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 74–89.
- [2] BROWN, A. B., AND PATTERSON, D. A. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Technical Conference* (San Antonio, TX, June 2003).
- [3] HA, J., ROSSBACH, C. J., DAVIS, J. V., ROY, I., RAMADAN, H. E., PORTER, D. E., CHEN, D. L., AND WITCHEL, E. Improved error reporting for software that uses black-box components. In *Proceedings of the Conference on Programming Language Design and Implementation 2007* (San Diego, CA, 2007).
- [4] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of Computer Security* 6, 3 (1998), 151–180.
- [5] HOLLAND, D. A., JOSEPHSON, W., MAGOUTIS, K., SELTZER, M., STEIN, C., AND LIM, A. Research issues in no-futz computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001), pp. 106–110.
- [6] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 223–236.
- [7] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.
- [8] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, October 2007), pp. 237–250.
- [9] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 245–257.
- [10] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of Usenix Large Installation Systems Administration Conference* (October 2003), pp. 159–172.
- [11] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 77–90.
- [12] YUAN, C., LAO, N., WEN, J.-R., LI, J., ZHANG, Z., WANG, Y.-M., AND MA, W.-Y. Automated known problem diagnosis with event traces. In *Proceedings of EuroSys 2006* (Leuven, Belgium, 2006).

Diverse Replication for Single-Machine Byzantine-Fault Tolerance

Byung-Gon Chun[†], Petros Maniatis^{*}, Scott Shenker^{†‡}

[†]ICSI, ^{*}Intel Research Berkeley, [‡]UC Berkeley

Abstract

New single-machine environments are emerging from abundant computation available through multiple cores and secure virtualization. In this paper, we describe the research challenges and opportunities around diversified replication as a method to increase the Byzantine-fault tolerance (BFT) of single-machine servers to software attacks or errors. We then discuss the design space of BFT protocols enabled by these new environments.

1 Introduction

Current commodity computing architectures still offer increasing, abundant power, despite the pessimists' periodic proclamations that the underlying technology has reached its limits. Chip makers double the number of standard processing cores per chip with every semiconductor process generation, and some research groups already design programming models and system architectures for tens to thousands of cores per chip [2, 6]. Nevertheless, the dependability of applications running on this abundant power has not necessarily similarly improved. First, as more transistors are crammed into the same chip area, soft errors and undetected hardware defects increase in incidence; furthermore, as cycle-hungry software expands to absorb what hardware evolution provides it, its complexity also increases leading to more bugs, more malicious exploits of those bugs, and more application crashes.

In this paper, we revisit the classic idea of replication-based system reliability: run the same application (say, a database server) in multiple instances on the same powerful server, and perform some form of consensus on the results of each request across replicas, to mask out faults. Multi-processor systems as far back as NonStop [10] and TARGON [14] used this idea to increase reliability at the instruction or kernel level, and distributed replicated services using Paxos [26] or Castro and Liskov's PBFT [16] are increasingly practical [4, 18, 19, 25]. In all cases, some form of voting among replicas helps choose the right result, as long as no more than a maximum fraction of all replicas are faulty at any time. In this position paper, we focus on using replication in single-server systems—that is, collocating multiple instances of the replicated service on the same physical machine—to take advantage of per-machine cycle abundance for Byzantine-fault tolerance to software attacks or errors. Performance is not necessarily the major challenge: running multiple

replicas in their own virtual machines (VMs) within a single multi-core system is already feasible without incurring performance overheads [15]. Our goal is to provide single-server systems with an increase in *reliability* that is commensurate with current trends in computation power.

A fundamental challenge towards our goal is ensuring that replicas of a server fail independently. For instance, if the exact same bugs can be triggered by a malicious exploit in all instances of the code, replication will be an ineffective dependability tool: no amount of voting will be able to mask faults if all replicas exhibit the same fault. Two trends in current research and technology make us hopeful on this front. First, recent hardware and software advances lead to improved isolation among different execution domains: Intel's LaGrande platform, a.k.a. Trusted Execution Technology (TXT) [1] and AMD's Presidio (Secure Virtual Machine—SVM) [5] can isolate threads, processes, or virtual machines from each other, and similar advances such as SELinux and Singularity [21] improve on software-only isolation. Second, much work has been done to increase the diversity and predictability of runtime systems, both to expose bugs that might cause silent faults, and to prevent adversaries from reliably exploiting software bugs. In particular, recent research has diversified Guest OSes and rewritten binaries via randomizing address space layout, system call interfaces, instruction set numbering, event delivery, etc. [9, 13, 17, 22, 24, 28, 34].

In what follows, we hope to describe the research challenges and opportunities around *diversified replication* as a method to increase the Byzantine-fault tolerance of single-machine servers. We start by treating the execution environment itself, in terms of its isolation and diversity characteristics (Section 2). Then we describe the design choices available in adapting traditional Byzantine-fault tolerant replication to such a single-machine execution environment, presenting some potentially promising design points in the space (Section 3). Finally, we discuss the ecosystem around such a dependability approach, including dealing with core defects, soft errors [11] or hardware failures (Section 4). We conclude with related work and our research agenda.

2 Challenges in a BFT Server

The execution environment we consider uses BFT replicated state machines within a single server to tolerate software Byzantine faults. Traditionally, such replication

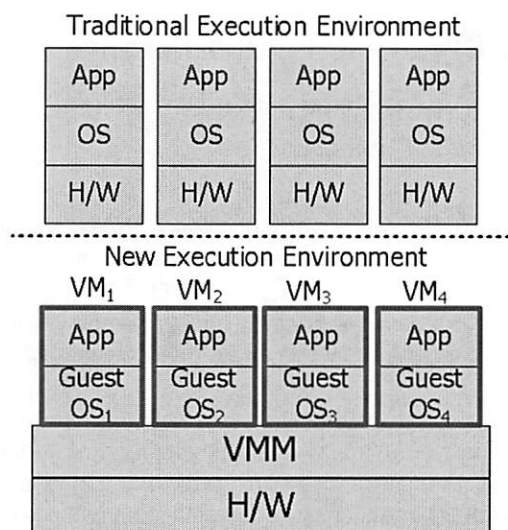


Figure 1: Traditional execution environment (top) vs. new execution environment (bottom). In our new execution environment, each replica runs in a VM running a diversified Guest OS.

is used across *multiple* machines, usually geographically distributed. There, a client issues an authenticated request to the replicas over the network. Those requests are ordered using methods such as agreement or ordered atomic broadcast. Then each replica executes each request in that order on its local copy of the service state. Figure 1(top) illustrates this traditional execution environment. Running multiple such replicas in a single machine is not obviously useful, since replicas compete with each other for resources slowing the whole system down, and they can interfere with each other spreading bugs or malicious faults, losing fault independence which is essential for replication to work. Thanks to multi-core architectures, the performance disincentives are no longer debilitating (especially for CPU-bound services).

To provide fault independence, the main challenges are *isolation* among replicas so that they do not interfere, and replica *diversity* so that they do not suffer from common vulnerabilities. We propose to virtualize replicas, running each replica in a VM with an automatically and systematically “diversified” Guest OS, replication software, and application replica (see Figure 1(bottom) for an illustration). Here, our trusted computing base (TCB) is the VMM and the hardware. Compared to making the entire OS and application stack reliable through sound software practices, making the VMM reliable enough to be justifiably inside the TCB is much more plausible: the Xen¹ VMM consists of tens of thousands of lines of code, but Microsoft Vista consists of 50 million lines of code even excluding the target application service.²

Next, we describe in detail what isolation features are

required for protecting domains (VMs) and how Guest OSes can be diversified automatically.

2.1 Isolation

To isolate protected domains (VMs), the VMM should provide isolation of computation, memory, and disk. In addition, it should provide a protected communication mechanism between VMs for efficient communication.

The VMM schedules CPU resources fairly among multiple VMs by partitioning CPU time. A VMM should ensure availability and liveness by not starving VMs. The VMM must protect physical memory pages of a VM from inappropriate accesses of other VMs. The VMM achieves this by partitioning the memory space, and controlling the paging mechanism by managing the register that contains the base address of the page directory. In addition, the VMM should shepherd DMA-capable devices.³ If a page is protected, the page is indicated in a secure table maintaining protected pages, a page fault occurs in a DMA, and the VMM disallows the DMA access to the page. Hardware trusted execution and protection such as Intel TXT [1] and AMD SVM [5] support a way to implement memory protection from DMA. For example, TXT supports the noDMA table that maintains this information at a chipset component.

The VMM also isolates virtual disks used by VMs by partitioning physical disk(s) into non-overlapping disk space. All disk I/Os are mediated by the VMM, thus a VM cannot directly access the virtual disk of other VMs. Finally, the VMM should ensure that communication between two VMs cannot be tampered with by another VM. No direct communication occurs among VMs, but the VMM mediates all communication among VMs.

There could be several ways of providing this isolation. The minimal mechanism that provides such isolation is a fundamental research question, especially in view of existing and future trusted hardware extensions for protected execution.

2.2 Diversity

Providing isolation is not enough to defend against software attacks. If replicas can be exploited simultaneously due to their common vulnerabilities, the replicated system cannot meet the bounded-fault assumption on which replication guarantees are founded. To avoid such correlated faults, replicas need to be diversified. Replicas are *diverse* if they maintain the same application semantics (i.e., they produce the same output given the same input) even though their implementation details (i.e., actual instructions executed) may be different.⁴ It is hard to compromise diverse replicas simultaneously via bug exploits or other such software attacks, since those attacks typically rely on specific memory layouts or instruction sequences.

Traditional approaches to achieving diversity include N -version programming [8] or using N different implementations of the same specification (e.g., by different vendors) [30–32]. N -version programming takes a design diversity approach: multiple teams produce different implementations of the same specification. N -version programming is rarely used in practice due to its high development and maintenance costs. Instead, opportunistically using existing independent implementations of standard services or interfaces has been more productive in practice. HACQIT [30] uses Apache running on Linux and IIS running on Microsoft Windows. HRDB [32] uses different DBMS implementations that support SQL. BASE [31] supports different NFS implementations through abstraction. Such implementation diversity is applicable only when different implementations that produce identical outputs exist and often when the service has a standardized high-level abstraction (e.g., SQL).

To create diversified replicas, we take a more broadly applicable approach, using OS and binary *randomization*. In particular, we propose to combine multiple existing randomization techniques to create diversified Guest OSes and/or binaries to run within VMs. Such randomization was used before to isolate bugs or to forestall particular intrusions via a specific attack vector. We use only techniques that preserve the same application semantics, so that it is safe to use voting over the replicas' results.

Combining diversification techniques is not straightforward. Some techniques can be used continuously, while some may conflict with each other, or in combination lead to unacceptably high overheads. We discuss some instances and lay out the challenges of creating diverse replicas.

Randomizing the location of stack or heap memory [13,22,34] has been explored to defend against buffer overflow attacks. Recent OSes such as Microsoft Vista and Mac OS X Leopard employ address space layout randomization (ASLR) as well. Randomizing interface mappings has also been studied: system call interface randomization changes mappings between system call numbers and code [17], while instruction set randomization changes mappings between opcodes and instructions. In both cases, an exploit meant to piggy-back executable code in a buffer overflow cannot know which system call number or even instruction opcode to use. Recovering from bugs in Rx [28] also relies on various randomizations: memory management randomizations including delayed recycling of freed buffers, padding allocated memory blocks, allocating memory in an alternate location, and zero-filling newly allocated memory buffers; and asynchronous event delivery including scheduling of events, signal delivery, and message re-ordering.

With these techniques, we can create diversified Guest OSes by combining randomization techniques with the following design goals: 1) maximizing the diversity of the replicas, 2) avoiding conflicting diversity techniques, and 3) controlling the overhead of diversity techniques.

There are a few research challenges in this scheme due to the use of randomization. First, among the sets of diversified Guest OSes, how does one choose those with the highest diversity? Naïvely using all randomization techniques together may be ineffective (e.g., delayed freeing combined with relocation defaults to one or the other according to their order of application). Even worse, some techniques in combination may result in conflicts or incur overheads beyond what applications can tolerate. For example, using two different stack randomization techniques at the same time may incur incorrect stack frames. The interplay of different types of techniques must be carefully mapped.

Creating diverse replicas using randomization also offers an opportunity to quantify the overall system probabilistically. We conjecture that increasing the number of replicas is likely to increase the safety of the system exponentially. To answer these questions, we would like to research a formal way to measure diversity and run real experiments with various attacks to quantify diversity.

Second, by using randomization techniques we may transform Byzantine faults to crash faults (e.g., because of failed exploits accessing disallowed memory regions). Exploits that were successful for the original undiversified code may translate to exploits that are still successful (i.e., cause safety violations), cause crashes, or are masked by our system. We need to examine how different exploits translate to different outcomes in our environment; e.g., this environment might have more crash faults than the original code. Having crash faults is clearly better than having faults violating safety, and we speculate that restarting VMs running replicas with crash faults in our secure virtualization environment can fix this problem without side effects. This is related to the Nysiad approach [23] of making crash-fault tolerant distributed applications to ones that are Byzantine-fault tolerant.

3 New BFT Opportunities

In this section, we discuss the spectrum of BFT replicated algorithms enabled by our new execution environment. In particular, the environment allows us to explore three axes of designing BFT systems: the trust characteristics of a coordinator process, the synchrony assumptions of communication among replicas, and the level of replication transparency towards clients.

The first design axis concerns whether the facility that orders client requests is trusted or not. In traditional replicated state machines, a replica acting as a *coordi-*

nator or primary typically assigns sequence numbers to requests. In a single-machine environment, this task can be taken on by a distinguished coordinator component. Since this functionality is simple, the coordinator may be straightforward to implement and formally prove correct, therefore justifying its inclusion in the TCB. On the other hand, not trusting the coordinator keeps the TCB leaner, but results in more complex replication protocols.

The second axis concerns the question of synchrony assumptions within the single machine. If one were to assume memory and I/O buses to be fault-free and all replicas equally fast regardless of diversification, then all communication among VMs can be thought of as synchronous. This simplifies the replication protocols, since no ambiguity exists among “slow” and maliciously “mute” replicas that try to stall the progress of the system. On the other hand, buses are not always fault-free, different diversification techniques or random seeds might result in vastly different execution times of the same code in different replicas at different times, making it difficult to maintain this synchrony assumption. Replication protocols for asynchronous environments are significantly more complex as a result. Bridging the gap between the two, one might imagine enforcing synchrony via the VMM by bounding execution time at replicas and restarting execution after a certain timeout, or resorting to eventual or virtual synchrony designs.

The final axis concerns the transparency of replication to clients. In traditional replicated systems, clients do interact with multiple replicas. In our setting, the coordinator can collect replies from multiple replicas inside a machine and interact with clients directly, offering the illusion of an unreplicated service, which is simpler and requires only a single communication session between the client and the server. In contrast, if replication is exposed to the client, individual replicas at the server communicate with the client directly. At the expense of greater bandwidth requirements and greater protocol complexity, exposure of replication removes the aggregation task (vote tallying, formation of a single response message) from the coordinator’s functionality.

In the design space we have described, there are many design combinations. For example, a design point that is closest to traditional agreement-based BFT protocols is built using an untrusted coordinator, asynchrony, and exposed replication. Here, we take two other unconventional design points from the design space and explain the resulting specific protocols in detail to illustrate that BFT can be achieved in a simpler way in our new execution environment.

To explain protocols concisely, we use the authentication notation of Yin et al. [35], according to which we denote by $\langle X \rangle_{S,D,k}$ an authentication certificate that any node in a set D can regard as proof that k distinct nodes

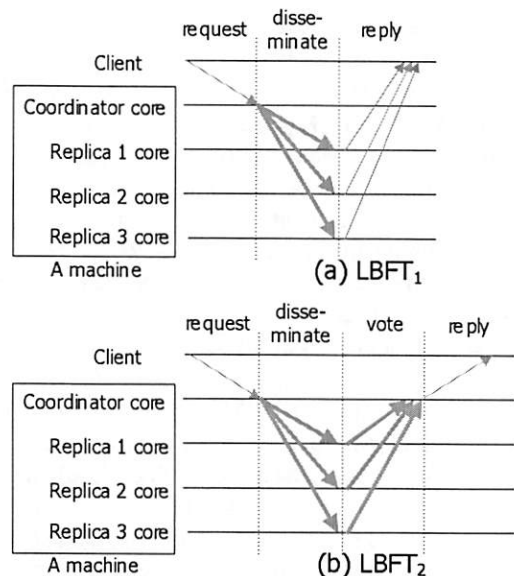


Figure 2: Lightweight BFT (LBFT) protocol instances. Thicker lines indicate communication inside a machine.

in S said X . For example, a traditional digital signature from p that is verifiable by the entire replica population R would be $\langle X \rangle_{\{p\},R,1}$, and a message authentication code (MAC) from p to q would be $\langle X \rangle_{\{p\},\{q\},1}$. If X is not signed, there is no subscript in the notation.

LBFT₁: In LBFT₁, we assume a trusted coordinator, asynchrony, and non-transparent replication. We run one coordinator and a set R of $2f + 1$ execution replicas to provide both safety and liveness with up to f Byzantine faults. Figure 2(a) shows LBFT₁, our simple protocol that uses the trusted coordinator. When p , the coordinator, receives $req = \langle REQUEST, o, t, c \rangle_{\{c\},R,1}$ from client c where o is the operation requested and t is the timestamp, it multicasts to R a $\langle DIST, n, req \rangle$ message where n is the assigned sequence number. Note that there is no authenticator in this message. Each replica executes the request, and sends a reply message to the client. When the client receives $f + 1$ valid matching reply messages forming the reply certificate $\langle REPLY, n, t, c, r \rangle_{R,\{c\},f+1}$, it accepts the result r .

It is worthwhile noting the distinct characteristics of this protocol. The coordinator does not deal with any cryptographic operation, but only replicas and clients perform the operations: there is no cryptographic operation between the coordinator and clients and between the coordinator and replicas. In this protocol, the client knows that it interacts with multiple replicas and performs majority voting from the received reply messages.

By trusting the coordinator, LBFT₁ reduces the complexity for implementing BFT protocols. It needs only a single phase request dissemination from the coordinator to the replicas with $O(N)$ messages. It does not need

to deal with a faulty coordinator (which would require complex *view change* algorithms). In addition, it needs $2f + 1$ replicas instead of $3f + 1$ replicas. The downside of the trusted coordinator is that our TCB grows slightly.

LBFT₂: By changing LBFT₁ slightly, we can create a protocol LBFT₂ that performs transparent replication. LBFT₂ (shown in Figure 2(b)) provides clients with an interface that is similar to interacting with a single server.

In LBFT₂, when p , the coordinator, receives a request message $\langle \text{REQUEST}, o, t, c \rangle_{\{c\}, \{p\}, 1}$, it checks whether the message authenticator is correct. If the authenticator is correct, it disseminates the message as in LBFT₁. Otherwise, it drops the message. When each replica executes the request, it sends its reply $\langle \text{REPLY}, n, t, c, r \rangle$ back to p . Note that there is no authenticator in the message. When p collects $f + 1$ valid matching reply messages (i.e., performs majority voting), it forwards to the client a reply message $\langle \text{REPLY}, n, t, c, r \rangle_{\{p\}, \{c\}, 1}$.

Unlike LBFT₁, this protocol incurs more cryptographic overhead at the coordinator, which must verify request authentication and collate/sign responses to clients. However, this protocol simplifies the interface to clients and reduces wide-area bandwidth overheads. Clearly many trade-offs exist in the customization of such replication protocols and different choices make sense for any given operating environment.

4 Discussion

Core failures: Core failures such as manufacturing defects and soft errors are becoming a concern as more and more transistors are put into a processor without a corresponding reliability increase for each individual transistor. We can mask these core failures by modifying the VMM running our architecture. First, the VMM should provide the capability to pin a VM at a specific core. If a VM is allowed to run at any core (typical in the symmetric multi-processing (SMP) model), then a defective core can affect all VMs, thus violating our fault assumptions. Second, the VMM itself should be able to handle core failures. This can be achieved by replicating VMM state, or by running the VMM at a more reliable core if such core heterogeneity is supported by the platform.

Machine crashes: Our BFT system runs in a single machine. If the machine crashes (e.g., hardware and power failure), our system is not available. To handle this type of failures, we can run our BFT system in multiple machines and coordinate the BFT groups. For example, an extended system coordinates BFT groups (one group per machine) using a benign fault tolerant replicated state machine algorithm like primary-backup replication and Paxos; this is a hybrid of benign and Byzantine fault tolerant protocols. There are other ways to organize these groups that give different fault tolerance properties and protocol complexity.

Adaptive replication: This replication utilizes idle cores opportunistically for improved fault tolerance. Instead of fixing the number of replicas, we can adaptively change the number of replicas based on load to keep performance the same. Adaptive replication trades off fault tolerance for performance.

Confidentiality: Our BFT system increases integrity and availability. However, the system can weaken confidentiality since compromising any one replica can leak sensitive information. This problem can be addressed by using threshold signatures [29] or privacy firewalls [35]. For example, to obtain sensitive information in threshold signature schemes, the attacker should compromise the *majority* of replicas.

5 Related Work

We briefly discuss other related work which is not presented in previous sections.

Diversity has also been explored for detecting *specific* malicious behavior. TightLip [36] runs a shadow process to detect the use of sensitive information. N-Variant Systems [20] aim to create general, *deterministic* variants with disjoint exploitation sets. The authors showed address space partitioning that detects attacks using absolute addresses and instruction set tagging. We might be able to borrow techniques from either camp to diversify virtualized applications on the fly.

Processor-level replication that addresses benign faults has been widely studied. All instructions of the application are replicated and checked through hardware redundancy [12, 33] or time redundancy [7]. These instruction-level replication schemes are suitable for masking hardware faults, but not for defending against software attacks. Executing the same instruction stream having common vulnerabilities multiple times does not help to mitigate Byzantine faults.

Replication systems such as MARE (Multiple Almost-Redundant Executions) [37] focus on reducing replication costs for software and configuration testing and checking. Such a partial-replication system executes a single instruction stream most of the time, and only runs redundant streams of instructions for some parts of a program. This approach could be used to provide the adaptive replication mechanism we envision in Section 4. Replicant [27] creates replicated processes with OS support and focuses on handling non-determinism caused by multiple threads running in a process.

6 Conclusion

In this paper, we argue for a new single-machine execution environment for BFT which is emerging from abundant computation through many cores, virtualization, and trusted execution and protection. To avoid common exploits, we propose to run diversified Guest OSes

in VMs by combining several OS and binary randomization techniques. In this new execution environment, we discuss new BFT protocol design opportunities, e.g., factoring out serialization from traditional BFT protocols and pushing it to our TCB, or designing protocols on synchrony assumptions. As future work, we plan to develop prototypes of systems from our design space and evaluate the diversity, availability, and performance of the systems.

References

- [1] Intel trusted execution technology. <http://www.intel.com/technology/security>.
- [2] Tera-scale computing research program. <http://techresearch.intel.com/articles/Tera-Scale/1421.htm>.
- [3] Xen. <http://xen.org>.
- [4] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.
- [5] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. 2005.
- [6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, UC Berkeley, 2006.
- [7] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Symposium on Microarchitecture*, 2001.
- [8] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault-tolerance during program execution. In *International Computer Software and Applications Conference*, 1977.
- [9] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM CCS*, 2003.
- [10] J. F. Bartlett. A NonStop kernel. In *SOSP*, 1981.
- [11] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE Reliability Physics Tutorial Notes, Reliability Fundamentals*, 2002.
- [12] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *DSN*, 2005.
- [13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, 2003.
- [14] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Trans. Comput. Syst.*, 7(1), 1989.
- [15] R. E. Carpenter. Comparing Multi-Core Processors for Server Virtualization. *IT@Intel*, Aug. 2007. http://www.intel.com/it/pdf/multicore_virtualization.pdf.
- [16] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4), 2002.
- [17] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. In *Technical Report CMU-CS-02-197, CMU*, 2002.
- [18] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *SOSP*, 2007.
- [19] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.
- [20] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security*, 2006.
- [21] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys*, 2006.
- [22] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HotOS*, 1997.
- [23] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. In *NSDI*, 2008.
- [24] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM CCS*, 2003.
- [25] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.
- [26] L. Lamport. The part-time parliament. In *ACM TOCS*, 1998.
- [27] J. Pool, I. S. K. Wong, and D. Lie. Relaxed determinism: Making redundant execution on multiprocessors practical. In *HotOS*, 2007.
- [28] F. Qin, J. Tucek, Jagadeesan, Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies – a safe method for surviving software failures. In *SOSP*, 2005.
- [29] T. Rabin. A simplified approach to threshold and proactive RSA. In *CRYPTO*, 1998.
- [30] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt. The design and implementation of an intrusion tolerant system. In *Foundations of Intrusion Tolerant Systems*, 2003.
- [31] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *SOSP*, 2001.
- [32] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine Faults in Database Systems using Commit Barrier Scheduling. In *SOSP*, 2007.
- [33] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *DSN*, 2001.
- [34] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In *SRDS*, 2003.
- [35] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP*, 2003.
- [36] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *NSDI*, 2007.
- [37] Y. Zhou, D. Marinov, W. Sanders, C. Zilles, M. d'Amorim, S. Lauterburg, R. M. Lefever, and J. Tucek. Delta execution for software reliability. In *HotDep*, 2007.

Notes

¹Xen [3] runs a special management VM (called Dom0) that hosts device drivers. To improve the fault tolerance of Dom0, Dom0 can also be replicated.

²Note that other possible architectures exist, e.g., using an unvirtualized secure operating system such as SELinux or Singularity, and running process-level diversification to the application replicas within regular isolated processes. Here we concentrate on a virtualization-based approach, for concreteness.

³DMA-capable devices are hardware, but they are typically configured by software.

⁴Of course, this works only if the replicas implement the *correct* application semantics; there is no amount of diversification that can make an application that implements the wrong protocol appear to implement the right protocol.

Vx32: Lightweight User-level Sandboxing on the x86

Bryan Ford and Russ Cox
Massachusetts Institute of Technology
{baford,rsc}@pdos.csail.mit.edu

Abstract

Code sandboxing is useful for many purposes, but most sandboxing techniques require kernel modifications, do not completely isolate guest code, or incur substantial performance costs. Vx32 is a multipurpose user-level sandbox that enables any application to load and safely execute one or more guest plug-ins, confining each guest to a system call API controlled by the host application and to a restricted memory region within the host's address space. Vx32 runs guest code efficiently on several widespread operating systems without kernel extensions or special privileges; it protects the host program from both reads and writes by its guests; and it allows the host to restrict the instruction set available to guests. The key to vx32's combination of portability, flexibility, and efficiency is its use of x86 segmentation hardware to sandbox the guest's data accesses, along with a lightweight instruction translator to sandbox guest instructions.

We evaluate vx32 using microbenchmarks and whole system benchmarks, and we examine four applications based on vx32: an archival storage system, an extensible public-key infrastructure, an experimental user-level operating system running atop another host OS, and a Linux system call jail. The first three applications export custom APIs independent of the host OS to their guests, making their plug-ins binary-portable across host systems. Compute-intensive workloads for the first two applications exhibit between a 30% slowdown and a 30% *speedup* on vx32 relative to native execution; speedups result from vx32's instruction translator improving the cache locality of guest code. The experimental user-level operating system allows the use of the guest OS's applications alongside the host's native applications and runs faster than whole-system virtual machine monitors such as VMware and QEMU. The Linux system call jail incurs up to 80% overhead but requires no kernel modifications and is delegation-based, avoiding concurrency vulnerabilities present in other interposition mechanisms.

1 Introduction

A *sandbox* is a mechanism by which a *host* software system may execute arbitrary *guest* code in a confined environment, so that the guest code cannot compromise or affect the host other than according to a well-defined policy. Sandboxing is useful for many purposes, such as running untrusted Web applets within a browser [6], safely extending operating system kernels [5, 32], and limiting potential damage caused by compromised applications [19, 22]. Most sandboxing mechanisms, however, either require guest code to be (re-)written in a type-safe language [5, 6], depend on special OS-specific facilities [8, 15, 18, 19], allow guest code unrestricted read access to the host's state [29, 42], or entail a substantial performance cost [33, 34, 37].

Vx32 is a lightweight sandbox for the x86 architecture that enables applications to run untrusted code efficiently on standard operating systems without requiring special privileges or kernel extensions. The vx32 sandbox runs standard x86 instructions, so guest code may be written in any language including assembly language, and may use advanced processor features such as vector (SSE) instructions. An application may host multiple sandbox instances at once; vx32 gives each guest its own dynamically movable and resizable address space within the host's space. Vx32 confines both guest reads and guest writes to the guest's designated address region in the host, protecting both the host's integrity and the privacy of any sensitive data (e.g., SSL keys) it may hold in its address space. Vx32 confines each guest's system calls to an API completely determined by the host application. The guest system call API need not have any relationship to that of the host operating system, so the host application can keep its guest environments independent of and portable across host operating systems.

The key to vx32's combination of flexibility and efficiency is to use different mechanisms to sandbox data accesses and instruction execution. Vx32 sandboxes guest

data accesses using the x86 processor's segmentation hardware, by loading a special data segment into the `ds`, `es`, and `ss` registers before executing guest code. Accessing data through this segment automatically confines both reads and writes to the guest's designated address region, with no performance overhead since the processor always performs segment translation anyway.

Since the vx32 sandbox runs entirely in user mode, however, vx32 cannot rely on the processor's privilege level mechanism to prevent the guest from escaping its sandbox—for example, the x86 privilege levels alone would not prevent the guest from changing the segment registers. Vx32 therefore prevents guest code from executing “unsafe” instructions such as segment register loads by using dynamic instruction translation [9, 34], rewriting each guest code sequence into a “safe” form before executing it. This dynamic translation incurs some performance penalty, especially on control flow instructions, which vx32 must rewrite to keep execution confined to its cache of safe, rewritten code. Since vx32 confines data accesses via segmentation, it does not need to rewrite most computation instructions, leaving safe code sequences as compact and efficient as the guest's original code. Vx32's on-demand translation can in fact improve the cache locality of the guest code, sometimes resulting in better performance than the original code, as seen previously in dynamic optimization systems [4].

Because common OS kernels already provide user-level access to the x86 segmentation hardware, vx32 does not require any special privileges or kernel extensions in order to fully sandbox all memory reads and writes that guest code performs.

Vx32 is implemented as a library that runs on Linux, FreeBSD, and Mac OS X and is being used in several applications. VXA [13] is an archival storage system that stores executable decoders along with compressed content in archives, using vx32 to run these decoders at extraction time; thus the archives are “self-extracting” but also safe and OS-independent. Alpaca [24] is an extensible PKI framework based on proof-carrying authorization [3] that uses vx32 to execute cryptographic algorithms such as SHA-1 [12] that form components of untrusted PKI extensions. Plan 9 VX is a port of the Plan 9 operating system [35] to user space: Plan 9 kernel code runs as a user-level process atop another OS, and unmodified Plan 9 user applications run under the Plan 9 kernel's control inside vx32. Vxlinux is a delegation-based system call interposition tool for Linux. All of these applications rely on vx32 to provide near-native performance: if an extension mechanism incurs substantial slowdown, then in practice most users will forego extensibility in favor of faster but less flexible schemes.

Previous papers on VXA [13] and Alpaca [24] briefly introduced and evaluated vx32 in the context of those ap-

plications. This paper focuses on the vx32 virtual machine itself, describing its sandboxing technique in detail and analyzing its performance over a variety of applications, host operating systems, and hardware. On real applications, vx32 consistently executes guest code within a factor of two of native performance; often the overhead is just a few percent.

This paper first describes background and related work in Section 2, then presents the design of vx32 in Section 3. Section 4 evaluates vx32 on its own, then Section 5 evaluates vx32 in the context of the above four applications, and Section 6 concludes.

2 Related Work

Many experimental operating system architectures permit one user process to isolate and confine others to enforce a “principle of least privilege”: examples include capability systems [25], L3's clan/chief model [26], Fluke's nested process architecture [14], and generic software wrappers [15]. The primary performance cost of kernel-mediated sandboxes like these is that of traversing hardware protection domains, though with careful design this cost can be minimized [27]. Other systems permit the kernel itself to be extended with untrusted code, via domain-specific languages [31], type-safe languages [5], proof-carrying code [32], or special kernel-space protection mechanisms [40]. The main challenge in all of these approaches is deploying a new operating system architecture and migrating applications to it.

Other work has retrofitted existing kernels with sandboxing mechanisms for user processes, even taking advantage of x86 segments much as vx32 does [8]. These mechanisms still require kernel modifications, however, which are not easily portable even between different x86-based OSes. In contrast, vx32 operates entirely in user space and is easily portable to any operating system that provides standard features described in Section 3.

System call interposition, a sandboxing method implemented by Janus [19] and similar systems [7, 17, 18, 22, 36], requires minor modifications to existing kernels to provide a means for one user process to filter or handle selected system calls made by another process. Since the sandboxed process's system calls are still fielded by the host OS before being redirected to the user-level “supervisor” process, system call interposition assumes that the sandboxed process uses the same basic system call API as the host OS: the supervisor process cannot efficiently export a completely different (e.g., OS-independent) API to the sandboxed process as a vx32 host application can. Some system call interposition methods also have concurrency-related security vulnerabilities [16, 43], whose only clear solution is delegation-based interposition [17]. Although vx32 has other uses,

it can be used is to implement efficient delegation-based system call interposition, as described in Section 5.4.

Virtualization has been in use for decades for purposes such as sharing resources [10] and migrating applications to new operating systems [20]. Since the x86 architecture did not provide explicit support for virtualization until recently, x86-based virtual machines such as VMware [1] had to use dynamic instruction translation to run guest kernel code in an unprivileged environment while simulating the appearance of being run in privileged mode: the dynamic translator rewrites instructions that might reveal the current privilege level. Virtual machines usually do not translate user-mode guest code, relying instead on host kernel extensions to run user-mode guest code directly in a suitably constructed execution environment. As described in Section 5.3, vx32's dynamic translation can be used to construct virtual machines that need no host kernel extensions, at some performance cost.

Dynamic instruction translation is frequently used for purposes other than sandboxing, such as dynamic optimization [4], emulating other hardware platforms [9, 44] or code instrumentation and debugging [28, 34]. The latter two uses require much more complex code transformations than vx32 performs, with a correspondingly larger performance cost [37].

A software fault isolation (SFI) system [29, 42] statically transforms guest code, preprocessing it to create a specialized version in which it is easy for the verifier to check that all data write instructions write only to a designated “guest” address range, and that control transfer instructions branch only to “safe” code entrypoints. SFI originally assumed a RISC architecture [42], but PittSFIeld adapted SFI to the x86 architecture [29]. SFI's preprocessing eliminates the need for dynamic instruction translation at runtime but increases program code size: e.g., 60%-100% for PittSFIeld. For efficiency, SFI implementations typically sandbox only writes and branches, not reads, so the guest can freely examine host code and data. This may be unacceptable if the host application holds sensitive data such as passwords or SSL keys. The main challenge in SFI on x86 is the architecture's variable-length instructions: opcode sequences representing unsafe instructions might appear in the middle of legitimate, safe instructions. PittSFIeld addresses this problem by inserting no-ops so that all branch targets are 16-byte aligned and then ensures that branches clear the bottom four bits of the target address. MiSFIT [39] sidesteps this problem for direct jumps by loading only code that was assembled and cryptographically signed by a trusted assembler. Indirect jumps consult a hash table listing valid jump targets.

Applications can use type-safe languages such as Java [6] or C# [30] to implement sandboxing completely in user space. This approach requires guest code to be

written in a particular language, making it difficult to reuse existing legacy code or use advanced processor features such as vector instructions (SSE) to improve the performance of compute-intensive code.

3 The Vx32 Virtual Machine

The vx32 virtual machine separates data sandboxing from code sandboxing, using different, complementary mechanisms for each: x86 segmentation hardware to sandbox data references and dynamic instruction translation to sandbox code. The dynamic instruction translation prevents malicious guest code from escaping the data sandbox. Vx32's dynamic translation is simple and lightweight, rewriting only indirect branches and replacing unsafe instructions with virtual traps. The use of dynamic translation also makes it possible for client libraries to restrict the instruction set further.

This section describes the requirements that vx32 places on its context—the processor, operating system, and guest code—and then explains the vx32 design.

3.1 Requirements

Processor architecture. Vx32 is designed around the x86 architecture, making the assumption that most systems now and in the foreseeable future are either x86-based or will be able to emulate x86 code efficiently. This assumption appears reasonable in the current desktop and server computing market, although it may prevent vx32 from spreading easily into other domains, such as game consoles and handheld mobile devices.

Vx32 uses protected-mode segmentation, which has been integral to the x86 architecture since before its extension to 32 bits [21]. The recent 64-bit extension of the architecture disables segment translation in 64-bit code, but still provides segmentation for 32-bit code [2]. Vx32 therefore cannot use segmentation-based data sandboxing to run 64-bit guest code, but it can still run 32-bit sandboxed guest code within a 64-bit host application.

Host operating system. Vx32 requires that the host OS provide a method of inserting custom segment descriptors into the application's local descriptor table (LDT), as explained below. The host OS can easily and safely provide this service to all applications, provided it checks and restricts the privileges of custom segments. All widely-used x86 operating systems have this feature.¹

To catch and isolate exceptions caused by guest code, vx32 needs to register its own signal handlers for processor exceptions such as segmentation faults and floating point exceptions. For full functionality and robustness, the host OS must allow vx32 to handle these signals on a

¹One Windows vulnerability, MS04-011, was caused by inadequate checks on application-provided LDT segments: this was merely a bug in the OS and not an issue with custom segments in general.

separate signal stack, passing vx32 the full saved register state when such a signal occurs. Again, all widely-used x86 operating systems have this capability.

Finally, vx32 can benefit from being able to map disk files into the host application's address space and to control the read/write/execute permissions on individual pages in the mapping. Although these features are not strictly required by vx32, they are, once again, provided by all widely-used x86 operating systems.

On modern Unix variants such as Linux, FreeBSD, and OS X, specific system calls satisfying the above requirements are `modify_ldt/i386_set_ldt`, `sigaction`, `sigaltstack`, `mmap`, and `mprotect`. Windows NT, 2000, and XP support equivalent system calls, though we have not ported vx32 to Windows. We have not examined whether Windows Vista retains this functionality.

Guest code. Although vx32 uses x86 segmentation for data sandboxing, it assumes that guest code running in the sandbox conforms to the 32-bit "flat model" and makes no explicit reference to segment registers. In fact, vx32 rewrites any guest instructions referring to segment registers so that they raise a virtual illegal instruction exception. This "flat model" assumption is reasonable for practically all modern, compiled 32-bit x86 code; it would typically be a problem only if, for example, the sandboxed guest wished to run 16-bit DOS or Windows code or wished to run a nested instance of vx32 itself.

Some modern multithreading libraries use segment registers to provide quick access to thread-local storage (TLS); such libraries cannot be used in guest code under the current version of vx32, but this is not a fundamental limitation of the approach. Vx32 could be enhanced to allow guest code to create new segments using emulation techniques, perhaps at some performance cost.

Host applications may impose further restrictions on guest code through configuration flags that direct vx32 to reject specific classes of instructions. For example, for consistent behavior across processor implementations, the VXA archiver described in Section 5.1 disallows the non-deterministic 387 floating-point instructions, forcing applications to use deterministic SSE-based equivalents.

3.2 Data sandboxing: segmentation

In the x86 architecture, segmentation is an address translation step that the processor applies immediately before page translation. In addition to the eight general-purpose registers (GPRs) accessible in user mode, the processor provides six *segment registers*. During any memory access, the processor uses the value in one of these segment registers as an index into one of two segment translation tables, the *global descriptor table* (GDT) or *local descriptor table* (LDT). The GDT traditionally describes segments shared by all processes, while the LDT contains segments specific to a particular process. Upon

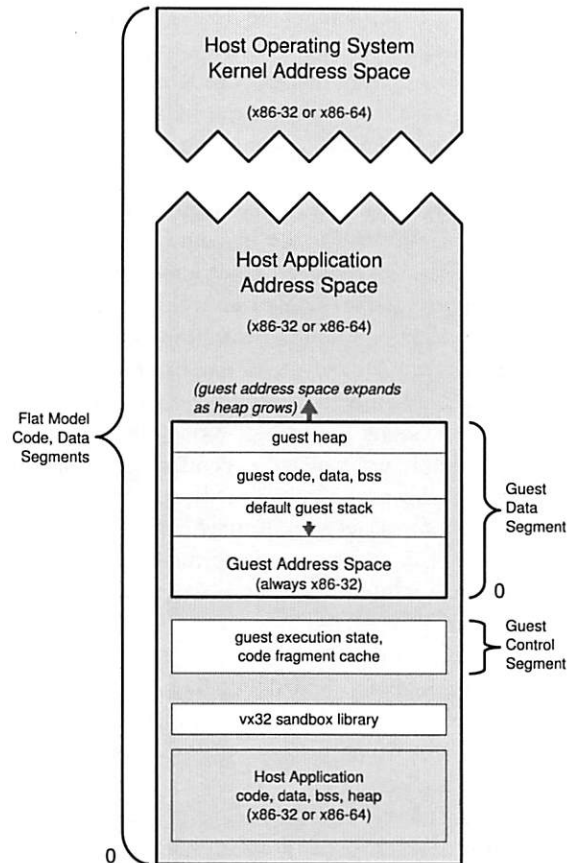


Figure 1: Guest and Host Address Space Structure

finding the appropriate descriptor table entry, the processor checks permission bits (read, write, and execute) and compares the virtual address of the requested memory access against the *segment limit* in the descriptor table, throwing an exception if any of these checks fail. Finally, the processor adds the *segment base* to the virtual address to form the *linear address* that it subsequently uses for page translation. Thus, a normal segment with base b and limit l permits memory accesses at virtual addresses between 0 and l , and maps these virtual addresses to linear addresses from b to $b+l$. Today's x86 operating systems typically make segmentation translation a no-op by using a base of 0 and a limit of $2^{32}-1$. Even in this so-called "flat model," the processor continues to perform segmentation translation: it cannot be disabled.

Vx32 allocates two segments in the host application's LDT for each guest instance: a *guest data segment* and a *guest control segment*, as depicted in Figure 1.

The *guest data segment* corresponds exactly to the guest instance's address space: the segment base points to the beginning of the address space (address 0 in the guest instance), and the segment size is the guest's address space size. Vx32 executes guest code with the processor's `ds`, `es`, and `ss` registers holding the selec-

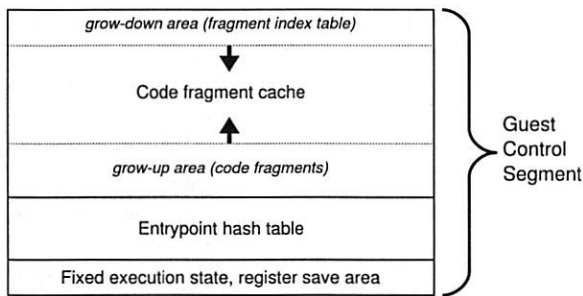


Figure 2: Guest Control Segment Structure

tor for the guest data segment, so that data reads and writes performed by the guest access this segment by default. (Code sandboxing, described below, ensures that guest code cannot override this default.) The segmentation hardware ensures that the address space appears at address 0 in the guest and that the guest cannot access addresses past the end of the segment. The translation also makes it possible for the host to unmap a guest's address space when it is not in use and remap it later at a different host address, to relieve congestion in the host's address space for example.

The format of the guest data segment is up to vx32's client: vx32 only requires that it be a contiguous, page-aligned range of virtual memory within the host address space. Vx32 provides a loader for ELF executables [41], but clients can load guests by other means. For example, Plan 9 VX (see section 5.3) uses `mmap` and `mprotect` to implement demand loading of Plan 9 executables.

The *guest control segment*, shown in Figure 2, contains the data needed by vx32 during guest execution. The segment begins with a fixed data structure containing saved host registers and other data. The *entrypoint hash table* and *code fragment cache* make up most of the segment. The hash table maps guest virtual addresses to code sequences in the code fragment cache. The translated code itself needs to be included in the guest control segment so that vx32 can write to it when patching previously-translated unconditional branches to jump directly to their targets [38].

Vx32 executes guest code with the processor's `fs` or `gs` register holding the selector for the guest control segment. The vx32 runtime accesses the control segment by specifying a segment override on its data access instructions. Whether vx32 uses `fs` or `gs` depends on the host system, as described in the next section.

3.3 Code sandboxing: dynamic translation

Data sandboxing ensures that, using the proper segments, data reads and writes cannot escape the guest's address space. Guests could still escape using segment override prefixes or segment register loads, however, which are unprivileged x86 operations. Vx32 therefore uses code

scanning and dynamic translation to prevent guest code from performing such unsafe operations.

As in Valgrind [34] and just-in-time compilation [11, 23], vx32's code scanning and translation is fully dynamic and runs on demand. The guest is allowed to place arbitrary code sequences in its address space, but vx32 never executes this potentially-unsafe code directly. Instead, whenever vx32 enters a guest instance, it translates a fragment of code starting at the guest's current instruction pointer (`eip`) to produce an equivalent safe fragment in vx32's code fragment cache, which lies *outside* the guest's address space. Vx32 also records the `eip` and address of the translated fragment in the entrypoint hash table for reuse if the guest branches to that `eip` again. Finally, vx32 jumps to the translated code fragment; after executing, the fragment either returns control to vx32 or jumps directly to the next translated fragment.

On 32-bit hosts, vx32 never changes the code segment register (`cs`): it jumps directly to the appropriate fragment in the guest's code fragment cache. This is safe because the code fragment cache only contains safe translations generated by vx32 itself. The code translator ensures that all branches inside translated code only jump to the beginning of other translated fragments or back to vx32 to handle events like indirect branches or virtualized guest system calls.

On 64-bit hosts, since segmentation only operates while executing 32-bit code, vx32 must create a special 32-bit code segment mapping the low 4GB of the host address space for use when running guest code. The guest control and data segments must therefore reside in the low 4GB of the host address space on such systems, although other host code and data may be above 4GB.

Because vx32 never executes code in the guest's address space directly, vx32 requires no static preprocessing or verification of guest code before it is loaded, in contrast with most other sandboxing techniques. Indeed, reliably performing static preprocessing and verification is problematic on the x86 due to the architecture's variable-length instructions [29, 39].

Translation overview. Vx32's translation of guest code into code fragments is a simple procedure with four stages: scan, simplify, place, and emit. The stages share a "hint table" containing information about each instruction in the fragment being translated. The eventual output is both the translated code and the hint table, which the translator saves for later use by exception handlers.

1. *Scan.* The translator first scans guest code starting at the desired `eip`, decoding x86 instructions to determine their lengths and any required transformations. The translator scans forward until it reaches an unconditional branch or a fragment size limit (currently about 128 bytes of instructions). The

scan phase records the length, original offset, instruction type, and worst-case translated size in the hint table. Jumps are the only instructions whose translated size is not known exactly at this point.

2. *Simplify.* The next phase scans the hint table for direct branches within the fragment being translated; it marks the ones that can be translated into short intrafragment branches using 8-bit jump offsets. After this phase, the hint table contains the exact size of the translation for each original guest instruction.
3. *Place.* Using the now-exact hint table information, the translator computes the exact offset of each instruction's translation. These offsets are needed to emit intrafragment branches in the last phase.
4. *Emit.* The final phase writes the translation into the code fragment cache. For most instructions, the translation is merely a copy of the original instruction; for "unsafe" guest instructions, the translation is an appropriate sequence chosen by vx32.

Vx32 saves the hint table, at a cost of four bytes per original instruction, in the code fragment cache alongside each translation, for use in exception handling as described in Section 3.4. The hint table could be discarded and recomputed during exception handling, trading exception handling performance for code cache space.

The rest of this section discusses specific types of guest instructions. Figure 3 shows concrete examples.

Computational code. Translation leaves most instructions intact. All ordinary computation and data access instructions (add, mov, and so on) and even floating-point and vector instructions are "safe" from vx32's perspective, requiring no translation, because the segmentation hardware checks all data reads and writes performed by these instructions against the guest data segment's limit. The only computation instructions that vx32 does not permit the guest to perform directly are those with x86 segment override prefixes, which change the segment register used to interpret memory addresses and could thus be used to escape the data sandbox.

Guest code may freely use all eight general-purpose registers provided by the x86 architecture: vx32 avoids both the dynamic register renaming and spilling of translation engines like Valgrind [34] and the static register usage restrictions of SFI [42]. Allowing guest code to use all the registers presents a practical challenge for vx32, however: it leaves no general-purpose register available where vx32 can store the address of the saved host registers for use while entering or exiting guest code. As mentioned above, vx32 solves this problem by placing the information in the guest control segment and using an otherwise-unused segment register (fs or gs) to address it. (Although vx32 does not permit segment

override prefixes in guest code, it is free to insert them for its own use in the code fragment translations.)

It is common nowadays for thread libraries to use one of these two segment registers—fs or gs—as a pointer to thread-local storage. If vx32 reused the thread-local segment register, it would have to restore the segment register before calling any thread-aware library routines, including routines that perform locking, such as printf. On recent GCC-based systems, the thread-local segment register is even used in function call prologues to look up the stack limit during a stack overflow check. Also, some 64-bit x86 operating systems (e.g., Linux) use privileged instructions to initialize the thread-local segment register with a base that is impossible to represent in an ordinary 32-bit segment descriptor. On such systems, restoring the thread-local segment register would require a system call, increasing the cost of exiting guest code. For these reasons, vx32 uses whichever segment register is not being used by the host OS's thread library. With care, vx32 could share the thread library's segment register.

Control transfers. To keep guest execution safely confined to its cache of translated code fragments, vx32 must ensure that all control transfer instructions—calls, jumps, and returns—go to vx32-generated translations, not to the original, unsafe guest code.

In the worst case, a control transfer must search the translation hash table, invoking the instruction translator if no translation exists. Once a translation has been found, vx32 can rewrite or "patch" direct jumps and direct calls to avoid future lookups [34, 38]. To implement this patching, the instruction translator initially translates each fixed-target jump or call instruction to jump to a stub that invokes the hash table lookup and branch patching function. The branch patching function looks up the target address and then rewrites the jump or call instruction to transfer directly to the target translation.

Patching cannot be used for indirect branches, including indirect calls and returns. This hash table lookup for indirect branches, especially during return instructions, is the main source of slowdown in vx32.

Other dynamic translation systems optimize indirect branches by caching the last target of each indirect branch and the corresponding translation address, or by maintaining a cache of subroutine return targets analogous to what many modern processors do [37]. Such optimizations would be unlikely to benefit vx32: its indirect target lookup path is only 21 instructions in the common case of an immediate hash table hit. Only the computation of the hash index—5 instructions—would be eliminated by using a single-entry branch cache. Most of the other instructions, which save and restore the x86 condition code flags and a few guest registers to give the target lookup code "room to work," would still be required no matter how simple the lookup itself.

(a) An indirect jump to the address stored at 08049248:

```

08048160  jmp     [0x08049248]
      ↓
b7d8d0f9  mov     ebx, fs:[0x2c]
b7d8d100  mov     fs:[0x2c], ebx
b7d8d107  mov     ebx, [0x08049248]
b7d8d10d  jmp     vxrun_lookup_indirect

```

The fs segment register points to the guest control segment. The first line of *every* translated code fragment is a prologue that restores the guest's ebx (at b7d8d0f9 in this case), because vx32 jumps into a fragment using a jmp [ebx] instruction.

The translation of the jmp instruction itself begins on the second line (at b7d8d100). The translated code saves ebx back into the guest control segment, loads the target eip into ebx, and then jumps to vxrun_lookup_indirect, which locates and jumps to the cached fragment for the guest address in ebx.

The first two lines cannot be optimized out: other fragments may directly jump past the first instruction, as shown below.

(b) A direct jump to 08048080:

```

08048160  jmp     0x08048080
      ↓
b7d8d0f9  mov     ebx, fs:[0x2c]
b7d8d100  jmp     0xb7d8d105
b7d8d105  mov     fs:[0x5c], 0x00008115
b7d8d110  jmp     vxrun_lookup_backpatch
b7d8d115  dword   0x08048080
b7d8d119  dword   0xb7d8d105

```

The first jmp in the translation is initially a no-op that just jumps to the next instruction, but vxrun_lookup_backpatch will rewrite it to avoid subsequent lookups. The word stored into fs:[0x5c] is an fs-relative offset telling vxrun_lookup_backpatch where in the control segment to find the two dwords arguments at b7d8d115. The control segment for the guest begins at b7d85000 in this example.

The first argument is the target eip; the second is the address of the end of the 32-bit jump offset to be patched. Since ebx has not been spilled at the point of the jump, vxrun_lookup_backpatch patches the jump to skip the one-instruction prologue in the target fragment that restores ebx.

(c) A return instruction:

```

08048160  ret
      ↓
b7d8d0f9  mov     ebx, fs:[0x2c]
b7d8d100  mov     fs:[0x2c], ebx
b7d8d107  pop     ebx
b7d8d108  jmp     vxrun_lookup_indirect

```

A return is an indirect jump to an address popped off the stack.

(d) An indirect call:

```

08048160  call    [0x08049248]
      ↓
b7d8d0f9  mov     ebx, fs:[0x2c]
b7d8d100  mov     fs:[0x2c], ebx
b7d8d107  mov     ebx, [0x08049248]
b7d8d10d  push    0x08048166
b7d8d112  jmp     vxrun_lookup_indirect

```

The translation is almost identical to the one in (a). The added push instruction saves the guest return address onto the stack.

(e) A direct call:

```

08048160  call    0x08048080
      ↓
b7d8d0f9  mov     ebx, fs:[0x2c]
b7d8d100  push    0x08048165
b7d8d105  jmp     0xb7d8d10a
b7d8d10a  mov     fs:[0x5c], 0x0000811a
b7d8d115  jmp     vxrun_lookup_backpatch
b7d8d11a  dword   0x08048080
b7d8d11e  dword   0xb7d8d10a

```

The translation is identical to the one in (b) except for the addition of the push that saves the return address.

(f) A software interrupt:

```

08048160  int     0x30
      ↓
b7d8d0f9  mov     ebx, fs:[0x2c]
b7d8d100  mov     fs:[0x20], eax
b7d8d106  mov     eax, 0x230
b7d8d10b  mov     fs:[0x40], 0x08048162
b7d8d116  jmp     vxrun_gentrap

```

The translation saves the guest eax into the guest control segment, loads the virtual trap number into eax (the 0x200 bit indicates an int instruction), saves the next eip into the guest control segment, and then jumps to the virtual trap handler, which will stop the execution loop and return from vx32, letting the library's caller handle the trap.

(g) An unsafe or illegal instruction:

```

08048160  mov     ds, ax
      ↓
b7d8d0f9  mov     ebx, fs:[0x2c]
b7d8d100  mov     fs:[0x20], eax
b7d8d106  mov     eax, 0x006
b7d8d10b  mov     fs:[0x40], 0x08048160
b7d8d116  jmp     vxrun_gentrap

```

The translation generates a virtual trap with code 0x006. In contrast with (f), for illegal instructions the saved eip points at the guest instruction itself rather than just past it.

Figure 3: Guest code and vx32 translations. Most instructions—arithmetic, data moves, and so on—are unchanged by translation.

Traps. Vx32 translates instructions like `int`, `syscall`, and `sysenter`, which normally generate hardware traps, into code sequences that generate virtual traps instead: they record the trap code and then cause vx32 to return to its caller, allowing the host application to handle the trap as it wishes. Typical applications look for a specific trap code to interpret as a “virtual system call” and treat any other trap as reason to terminate the guest.

Privileged or unsafe instructions. Vx32 translates privileged or unsafe instructions (for example, kernel-mode instructions or those user-mode instructions that manipulate the segment registers) into sequences that generate (virtual) illegal instruction traps.

3.4 Exception handling

With help from the host OS, vx32 catches processor exceptions in guest code—for example, segmentation violations and floating point exceptions—and turns them into virtual traps, returning control to the host application with full information about the exception that occurred.

Since the `eip` reported by the host OS on such an exception points into one of vx32’s code translations, vx32 must translate this `eip` back to the corresponding `eip` in the guest’s original instruction stream in order for it to make sense to the host application or the developer. To recover this information, vx32 first locates the translation fragment containing the current `eip` and converts the `eip`’s offset within the fragment to an offset from the guest code address corresponding to the fragment.

To locate the translation fragment containing the trapping `eip` efficiently, vx32 organizes the code fragment cache into two sections as shown earlier in Figure 2: the code translations and instruction offset tables are allocated from the bottom up, and the fragment index is allocated from the top down. The top-down portion of the cache is thus a table of all the translation fragments, sorted in reverse order by fragment address. The exception handler uses a binary search in this table to find the fragment containing a particular `eip` as well as the hint table constructed during translation.

Once vx32’s exception handler has located the correct fragment, it performs a second binary search, this one in the fragment’s hint table, to find the exact address of the guest instruction corresponding to the current `eip`.

Once the exception handler has translated the faulting `eip`, it can finally copy the other guest registers unchanged and exit the guest execution loop, transferring control back to the host application to handle the fault.

3.5 Usage

Vx32 is a generic virtual execution library; applications decide how to use it. Typically, applications use vx32 to execute guest code in a simple control loop: load a register set into the vx32 instance, and call vx32’s run

function; when run eventually returns a virtual trap code, handle the virtual trap; repeat. Diversity in vx32 applications arises from what meaning they assign to these traps. Section 5 describes a variety of vx32 applications and evaluates vx32 in those contexts.

Vx32 allows the creation of multiple guest contexts that can be run independently. In a multithreaded host application, different host threads can run different guest contexts simultaneously with no interference.

4 Vx32 Evaluation

This section evaluates vx32 in isolation, comparing vx32’s execution against native execution through microbenchmarks and whole-system benchmarks. Section 5 evaluates vx32 in the context of real applications. Both sections present experiments run on a variety of test machines, listed in Figure 4.

4.1 Implementation complexity

The vx32 sandbox library consists of 3,800 lines of C (1,500 semicolons) and 500 lines of x86 assembly language. The code translator makes up about half of the C code. Vx32 runs on Linux, FreeBSD, and Mac OS X without kernel modifications or access to privileged operating system features.

In addition to the library itself, the vx32 system provides a GNU compiler toolchain and a BSD-derived C library for optional use by guests hosted by applications that provide a Unix-like system call interface. Host applications are, of course, free to use their own compilers and libraries and to design new system call interfaces.

4.2 Microbenchmarks

To understand vx32’s performance costs, we wrote a small suite of microbenchmarks exercising illustrative cases. Figure 5 shows vx32’s performance on these tests.

Jump. This benchmark repeats a sequence of 100 no-op short jumps. Because a short jump is only two bytes, the targets are only aligned on 2-byte boundaries. In contrast, vx32’s generated fragments are aligned on 4-byte boundaries. The processors we tested vary in how sensitive they are to jump alignment, but almost all run considerably faster on vx32’s 4-byte aligned jumps than the 2-byte jumps in the native code. The Pentium 4 and the Xeon are unaffected.

Jumpal. This benchmark repeats a sequence of 100 short jumps that are spaced so that each jump target is aligned on a 16-byte boundary. Most processors execute vx32’s equivalent 4-byte aligned jumps a little slower. The Pentium 4 and Xeon are, again, unaffected.

Jumpfar. This benchmark repeats a sequence of 100 jumps spaced so that each jump target is aligned on a 4096-byte (page) boundary. This is a particularly hard

Label	CPU(s)	RAM	Operating System
Athlon64 x86-32	1.0GHz AMD Athlon64 2800+	2GB	Ubuntu 7.10, Linux 2.6.22 (32-bit)
Core 2 Duo	1x2 2.33GHz Intel Core 2 Duo	1GB	Mac OS X 10.4.10
Opteron x86-32	1.4GHz AMD Opteron 240	1GB	Ubuntu 7.10, Linux 2.6.22 (32-bit)
Opteron x86-64	1.4GHz AMD Opteron 240	1GB	Ubuntu 7.10, Linux 2.6.22 (64-bit)
Pentium 4	3.06GHz Intel Pentium 4	2GB	Ubuntu 7.10, Linux 2.6.22
Pentium M	1.0GHz Intel Pentium M	1GB	Ubuntu 7.04, Linux 2.6.10
Xeon	2x2 3.06GHz Intel Xeon	2GB	Debian 3.1, Linux 2.6.18

Figure 4: Systems used during vx32 evaluation. The two Opteron listings are a single machine running different operating systems. The notation 1x2 indicates a single-processor machine with two cores. All benchmarks used gcc 4.1.2.

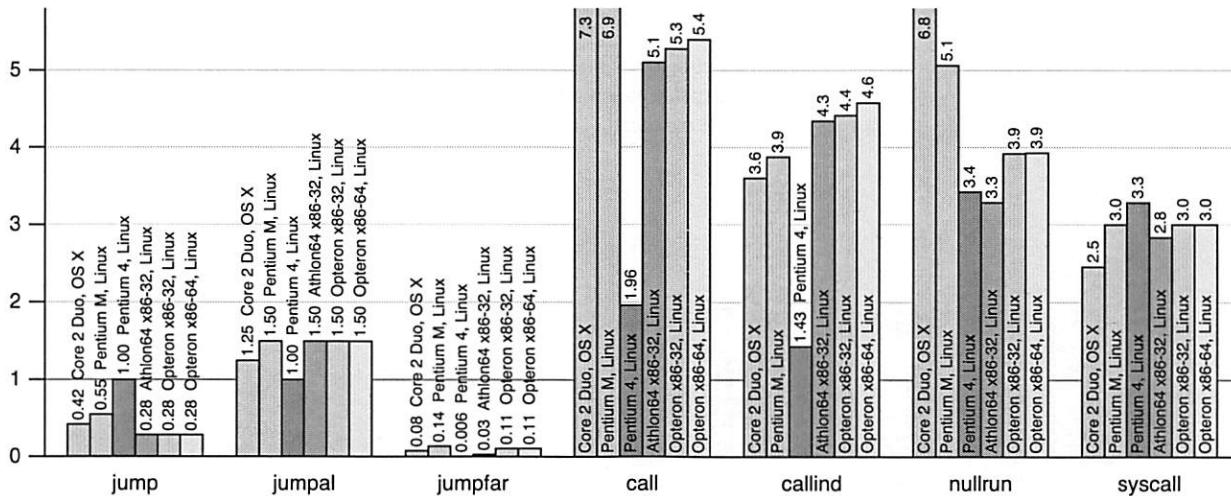


Figure 5: Normalized run times for microbenchmarks running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster vx32 runs). The benchmarks are described in Section 4.2. Results for the Intel Xeon matched the Pentium 4 almost exactly and are omitted for space reasons.

case for native execution, especially if the processor's instruction cache uses only the low 12 bits of the instruction address as the cache index. Vx32 runs this case significantly faster on all processors, because of better instruction cache performance in the translation.

Call. This benchmark repeatedly calls a function containing only a return instruction. The call is a direct branch, though the return is still an indirect branch.

Callind. This benchmark is the same as *call*, but the call is now an indirect branch, via a register.

Comparing the bars for *call* against the bars for *callind* may suggest that vx32 takes longer to execute direct function calls than indirect function calls, but only relative to the underlying hardware: a vx32 indirect call takes about twice as long as a vx32 direct call, while a native indirect call takes about four times as long as a native direct call. The *call* bars are taller than the *callind* bars not because vx32 executes direct calls more slowly, but because native hardware executes them so much faster.

Nullrun. This benchmark compares creating and executing a vx32 guest instance that immediately exits against forking a host process that immediately exits.

Syscall. This benchmark compares a virtual system call relayed to the host system against the same system call executed natively. (The system call is `close(-1)`, which should be trivial for the OS to execute.)

4.3 Large-scale benchmarks

The microbenchmarks help to characterize vx32's performance executing particular kinds of instructions, but the execution of real programs depends critically on how often the expensive instructions occur. To test vx32 on real programs, we wrote a 500-line host application called *vxrun* that loads ELF binaries [41] compiled for a generic Unix-like system call interface. The system call interface is complete enough to support the SPEC CPU2006 integer benchmark programs, which we ran both using vx32 (*vxrun*) and natively. We ran only the C integer benchmarks; we excluded 403.gcc and 429.mcf because they caused our test machines, most of which have only 1GB of RAM, to swap.

Figure 6 shows the performance of vx32 compared to the native system on five different 32-bit x86 processors. On three of the seven benchmarks, vx32 incurs a perfor-

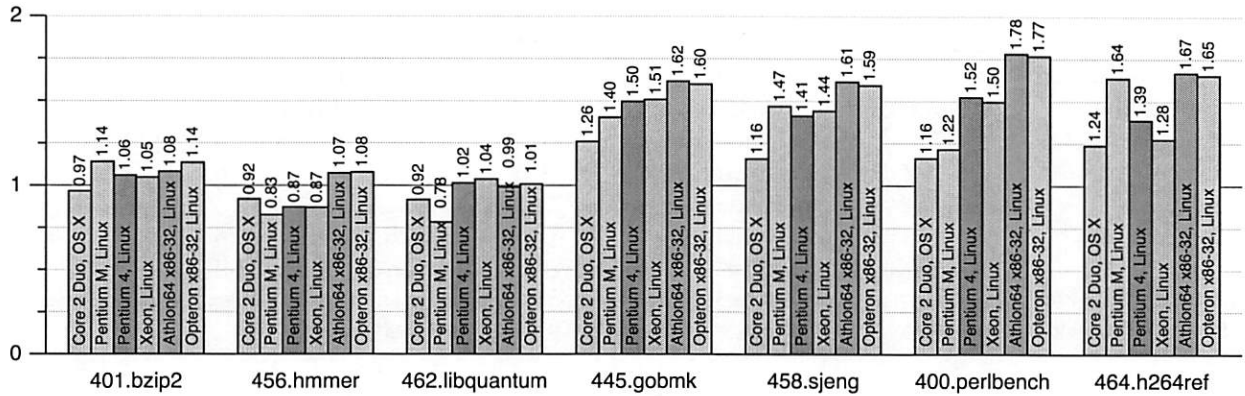


Figure 6: Normalized run times for SPEC CPU2006 benchmarks running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster vx32 runs). The left three benchmarks use fewer indirect branches than the right four, resulting in less vx32 overhead. The results are discussed further in Section 4.3.

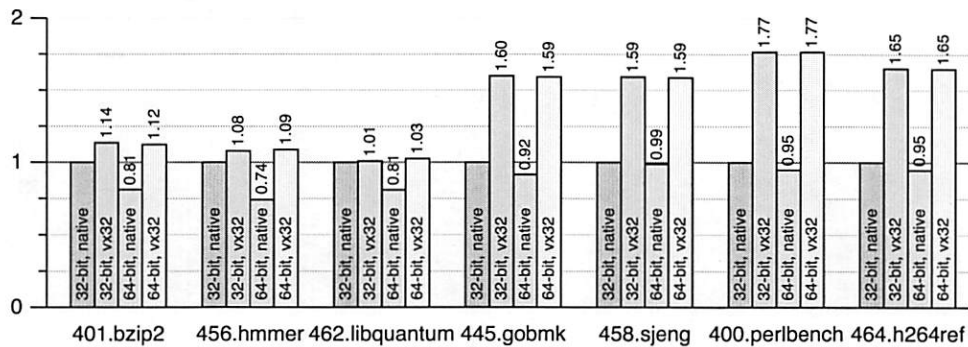


Figure 7: Normalized run times for SPEC CPU2006 benchmarks running in four configurations on the same AMD Opteron system: natively on 32-bit Linux, under vx32 hosted by 32-bit Linux, natively on 64-bit Linux, and under vx32 hosted by 64-bit Linux. Each bar plots run time divided by run time for the same benchmark running natively on 32-bit Linux (smaller bars mark faster runs). Vx32 performance is independent of the host operating system's choice of processor mode, because vx32 always runs guest code in 32-bit mode. The results are discussed further in Section 4.3.

mance penalty of less than 10%, yet on the other four, the penalty is 50% or more. The difference between these two groups is the relative frequency of indirect branches, which, as discussed in Section 3, are the most expensive kind of instruction that vx32 must handle.

Figure 8 shows the percentage of indirect branches retired by our Pentium 4 system during each SPEC benchmark, obtained via the CPU's performance counters [21]. The benchmarks that exhibit a high percentage of indirect call, jump, and return instructions are precisely those that suffer a high performance penalty under vx32.

We also examined vx32's performance running under a 32-bit host operating system compared to a 64-bit host operating system. Figure 7 graphs the results. Even under a 64-bit operating system, the processor switches to 32-bit mode when executing vx32's 32-bit code segments, so vx32's execution time is essentially identical in each case. Native 64-bit performance often differs from 32-bit performance, however: the x86-64 architecture's eight additional general-purpose registers can improve performance by requiring less register spilling in

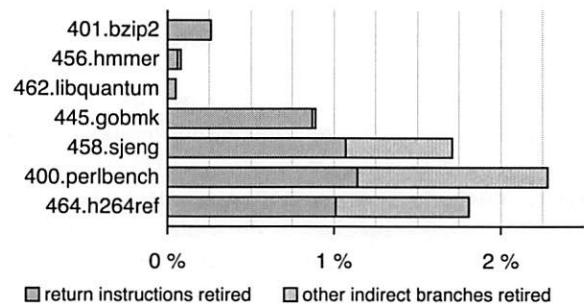


Figure 8: Indirect branches as a percentage of total instructions retired during SPEC CPU2006 benchmarks, measured using performance counters on the Pentium 4. The left portion of each bar corresponds to return instructions; the right portion corresponds to indirect jumps and indirect calls. The indirect-heavy workloads are exactly those that experience noticeable slowdowns under vx32.

compiled code, but its larger pointer size can hurt performance by decreasing cache locality, and the balance between these factors depends on the workload.

5 Applications

In addition to evaluating vx32 in isolation, we evaluated vx32 in the context of several applications built using it. This section evaluates the performance of these applications, but equally important is the ability to create them in the first place: vx32 makes it possible to create interesting new applications that execute untrusted x86 code on legacy operating systems without kernel modifications, at only a modest performance cost.

5.1 Archival storage

VXA [13] is an archival storage system that uses vx32 to “future proof” compressed data archives against changes in data compression formats. Data compression algorithms evolve much more rapidly than processor architectures, so VXA packages executable decoders into the compressed archives along with the compressed data itself. Unpacking the archive in the future then depends only on being able to run on (or simulate) an x86 processor, not on having the original codecs used to compress the data and being able to run them natively on the latest operating systems. Crucially, archival storage systems need to be efficiently usable now as well as in the future: if “future proofing” an archive using sandboxed decoders costs too much performance in the short term, the archive system is unlikely to be used except by professional archivists.

VXA uses vx32 to implement a minimal system call API (read, write, exit, sbrk). Vx32 provides exactly what the archiver needs: it protects the host from buggy or malicious archives, it isolates the decoders from the host’s system call API so that archives are portable across operating systems and OS versions, and it executes decoders efficiently enough that VXA can be used as a general-purpose archival storage system without noticeable slowdown. To ensure that VXA decoders behave identically on all platforms, VXA instructs vx32 to disable inexact instructions like the 387 intrinsics whose precise results vary from one processor to another; VXA decoders simply use SSE and math library equivalents.

Figure 9 shows the performance of vx32-based decoders compared to native ones on the four test architectures. All run within 30% of native performance, often much closer. The jpeg decoder is consistently faster under vx32 than natively, due to better cache locality.

5.2 Extensible public key infrastructure

Alpaca [24] is an extensible public-key infrastructure (PKI) and authorization framework built on the idea of proof-carrying authorization (PCA) [3], in which one party authenticates itself to another by using an explicit logical language to *prove* that it deserves a particular kind of access or is authorized to request particular ser-

vices. PCA systems before Alpaca assumed a fixed set of cryptographic algorithms, such as public-key encryption, signature, and hash algorithms. Alpaca moves these algorithms into the logical language itself, so that the extensibility of PCA extends not just to delegation policy but also to complete cryptographic suites and certificate formats. Unfortunately, cryptographic algorithms like round-based hash functions are inefficient to express and evaluate explicitly using Alpaca’s proof language.

Alpaca uses Python bindings for the vx32 sandbox to support native implementations of expensive algorithms like hashes, which run as untrusted “plug-ins” that are fully isolated from the host system. The lightweight sandboxing vx32 provides is again crucial to the application, because an extensible public-key infrastructure is unlikely to be used in practice if it makes all cryptographic operations orders of magnitude slower than native implementations would be.

Figure 10 shows the performance of vx32-based hash functions compared to native ones. All run within 25% of native performance. One surprise is the Core 2 Duo’s excellent performance, especially on whirlpool. We believe the Core 2 Duo is especially sensitive to cache locality.

5.3 Plan 9 VX

Plan 9 VX (9vx for short) is a port of the Plan 9 operating system [35] to run on top of commodity operating systems, allowing the use of both Plan 9 and the host system simultaneously and also avoiding the need to write hardware drivers. To run user programs, 9vx creates an appropriate address space in a window within its own address space and invokes vx32 to simulate user mode execution. Where a real kernel would execute `iret` to enter user mode and wait for the processor to trap back into kernel mode, 9vx invokes vx32 to simulate user mode, waiting for it to return with a virtual trap code. 9vx uses a temporary file as a simulation of physical memory, calling the host `mmap` and `mprotect` system calls to map individual memory pages as needed. This architecture makes it possible to simulate Plan 9’s shared-memory semantics exactly, so that standard Plan 9 x86 binaries run unmodified under 9vx. For example, Plan 9 threads have a shared address space except that each has a private stack. This behavior is foreign to other systems and very hard to simulate directly. Because all user-mode execution happens via vx32, 9vx can implement this easily with appropriate memory mappings.

The most surprising aspect of 9vx’s implementation was how few changes it required. Besides removing the hardware drivers, it required writing about 1,000 lines of code to interface with vx32, and another 500 to interface with the underlying host operating system. The changes mainly have to do with page faults. 9vx treats vx32 like an architecture with a software-managed TLB (the code

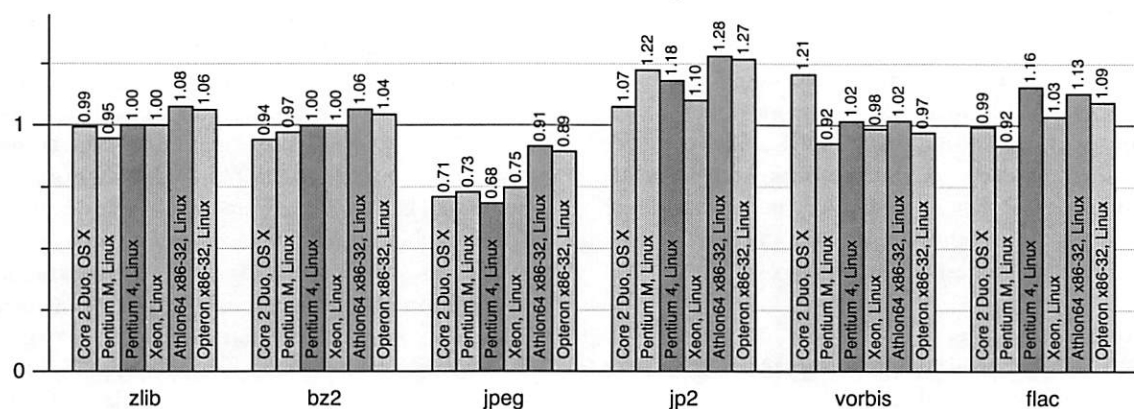


Figure 9: Normalized run times for VXA decoders running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster vx32 runs). Section 5.1 gives more details. The jpeg test runs faster because the vx32 translation has better cache locality than the original code.

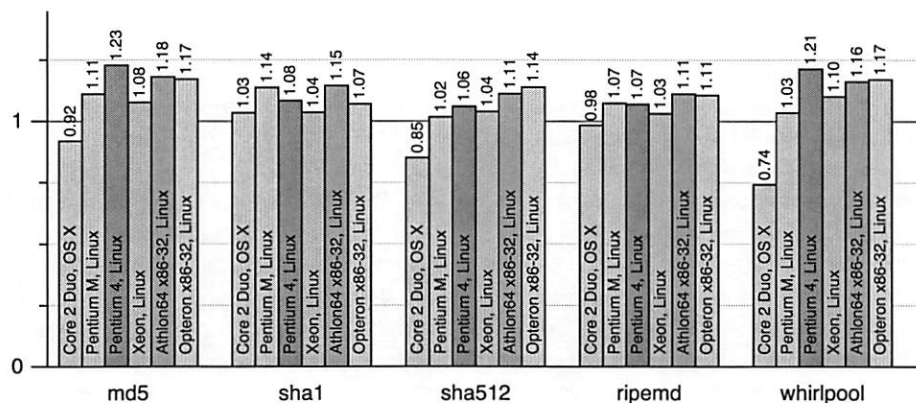


Figure 10: Normalized run times for cryptographic hash functions running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster runs).

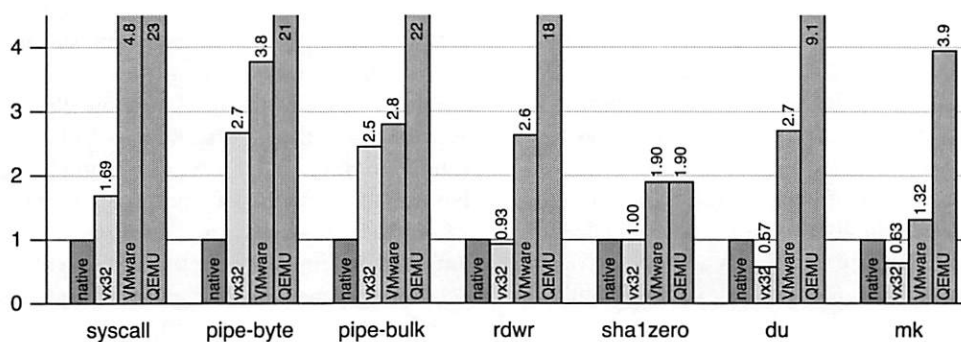


Figure 11: Normalized run times for simple Plan 9 benchmarks. The four bars correspond to Plan 9 running natively, Plan 9 VX, Plan 9 under VMware Workstation 6.0.2 on Linux, and Plan 9 under QEMU on Linux using the qemu kernel extension. Each bar plots run time divided by the native Plan 9 run time (smaller bars mark faster runs). The tests are: switch, a system call that reschedules the current process, causing a context switch (`sleep(0)`); pipe-byte, two processes sending a single byte back and forth over a pair of pipes; pipe-bulk, two processes (one sender, one receiver) transferring bulk data over a pipe; rdwr, a single process copying from `/dev/zero` to `/dev/null`; sha1zero, a single process reading `/dev/zero` and computing its SHA1 hash; du, a single process traversing the file system; and mk, building a Plan 9 kernel. See Section 5.3 for performance explanations.

was already present in Plan 9 to support architectures like the MIPS). 9vx unmaps all mapped pages during a process context switch (a single `munmap` call) and then remaps pages on demand during vx32 execution. A fault on a missing page causes the host kernel to send 9vx a signal (most often SIGSEGV), which causes vx32 to stop and return a virtual trap. 9vx handles the fault exactly as Plan 9 would and then passes control back to vx32. 9vx preempts user processes by asking the host OS to deliver SIGALRM signals at regular intervals; vx32 translates these signals into virtual clock interrupts.

To evaluate the performance of 9vx, we ran benchmarks on our Pentium M system in four configurations: native Plan 9, 9vx on Linux, Plan 9 under VMware Workstation 6.0.2 (build 59824) on Linux, and Plan 9 under QEMU on Linux with the `kqemu` module. Figure 11 shows the results. 9vx is slower than Plan 9 at context switching, so switch-heavy workloads suffer (`swtch`, `pipe-byte`, `pipe-bulk`). System calls that don't context switch (`rdwr`) and ordinary computation (`shlzero`) run at full speed under 9vx. In fact, 9vx's simulation of system calls is faster than VMware's and QEMU's, because it doesn't require simulating the processor's entry into and exit from kernel mode. File system access (`du`, `mk`) is also faster under 9vx than Plan 9, because 9vx uses Linux's in-kernel file system while the other setups use Plan 9's user-level file server. User-level file servers are particularly expensive in VMware and QEMU due to the extra context switches. We have not tested Plan 9 under VMware ESX server, which could be more efficient than VMware Workstation since it bypasses the host OS completely.

The new functionality 9vx creates is more important than its performance. Using vx32 means that 9vx requires no special kernel support to make it possible to run Plan 9 programs and native Unix programs side-by-side, sharing the same resources. This makes it easy to experiment with and use Plan 9's features while avoiding the need to maintain hardware drivers and port large pieces of software (such as web browsers) to Plan 9.

5.4 Vxlinux

We implemented a 250-line host application, vxlinux, that provides delegation-based interposition [17] by running unmodified, single-threaded Linux binaries under vx32 and relaying the guest's system calls to the host OS. A complete interposition system would include a policy controlling which system calls to relay, but for now we merely wish to evaluate the basic interposition mechanism. The benefit of vxlinux over the OS-independent vxrun (described in Section 4) is that it runs unmodified Linux binaries without requiring recompilation for vx32. The downside is that since it implements system calls by passing arguments through to the Linux kernel,

it can only run on Linux. The performance of the SPEC benchmarks under vxlinux is essentially the same as the performance under vxrun; we omit the graph.

6 Conclusion

Vx32 is a multipurpose user-level sandbox that enables any application to load and safely execute one or more guest plug-ins, confining each guest to a system call API controlled by the host application and to a restricted memory region within the host's address space. It executes sandboxed code efficiently on x86 architecture machines by using the x86's segmentation hardware to isolate memory accesses along with dynamic code translation to disallow unsafe instructions.

Vx32's ability to sandbox untrusted code efficiently has enabled a variety of interesting applications: self-extracting archival storage, extensible public-key infrastructure, a user-level operating system, and portable or restricted execution environments. Because vx32 works on widely-used x86 operating systems without kernel modifications, these applications are easy to deploy.

In the context of these applications (and also on the SPEC CPU2006 benchmark suite), vx32 always delivers sandboxed execution performance within a factor of two of native execution. Many programs execute within 10% of the performance of native execution, and some programs execute faster under vx32 than natively.

Acknowledgments

Chris Lesniewski-Laas is the primary author of Alpaca. We thank Austin Clements, Stephen McCamant, and the anonymous reviewers for valuable feedback. This research is sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan, and by the National Science Foundation under FIND project 0627065 (User Information Architecture).

References

- [1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASP-LOS XIII*, December 2006.
- [2] Advanced Micro Devices, Inc. AMD x86-64 architecture programmer's manual, September 2002.
- [3] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM CCS*, November 1999.
- [4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.
- [5] Brian N. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *15th SOSP*, 1995.

- [6] Brian Case. Implementing the Java virtual machine. *Microprocessor Report*, 10(4):12–17, March 1996.
- [7] Suresh N. Chari and Pau-Chen Cheng. BlueBox: A policy-driven, host-based intrusion detection system. In *Network and Distributed System Security*, February 2002.
- [8] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *17th SOSP*, pages 140–153, December 1999.
- [9] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *SIGMETRICS PER*, 22(1):128–137, May 1994.
- [10] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.
- [11] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Principles of Programming Languages*, pages 297–302, Salt Lake City, UT, January 1984.
- [12] D. Eastlake 3rd and T. Hansen. US secure hash algorithms (SHA and HMAC-SHA), July 2006. RFC 4634.
- [13] Bryan Ford. VXA: A virtual architecture for durable compressed archives. In *4th USENIX FAST*, San Francisco, CA, December 2005.
- [14] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.
- [15] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.
- [16] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Network and Distributed System Security*, February 2003.
- [17] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Network and Distributed System Security*, February 2004.
- [18] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX*, June 1998.
- [19] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *6th USENIX Security Symposium*, San Jose, CA, 1996.
- [20] Honeywell Inc. *GCOS Environment Simulator*. December 1983. Order Number AN05-02A.
- [21] Intel Corporation. IA-32 Intel architecture software developer’s manual, June 2005.
- [22] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Network and Distributed System Security*, February 2000.
- [23] Andreas Krall. Efficient JavaVM just-in-time compilation. In *Parallel Architectures and Compilation Techniques*, pages 54–61, Paris, France, October 1998.
- [24] Christopher Lesniewski-Laas, Bryan Ford, Jacob Strauss, M. Frans Kaashoek, and Robert Morris. Alpaca: extensible authorization for distributed services. In *ACM Computer and Communications Security*, October 2007.
- [25] Henry M Levy. *Capability-based Computer Systems*. Digital Press, 1984.
- [26] Jochen Liedtke. A persistent system in real use: experiences of the first 13 years. In *IWOOS*, 1993.
- [27] Jochen Liedtke. On micro-kernel construction. In *15th SOSP*, 1995.
- [28] Chi-Keung Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, June 2005.
- [29] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, August 2006.
- [30] Microsoft Corporation. C# language specification, version 3.0, 2007.
- [31] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Symposium on Operating System Principles*, pages 39–51, Austin, TX, November 1987.
- [32] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd OSDI*, pages 229–243, 1996.
- [33] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Third Workshop on Runtime Verification (RV’03)*, Boulder, CO, July 2003.
- [34] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, June 2007.
- [35] Rob Pike et al. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.
- [36] Niels Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, August 2003.
- [37] K. Scott et al. Overhead reduction techniques for software dynamic translation. In *NSF Workshop on Next Generation Software*, April 2004.
- [38] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.
- [39] Christopher Small and Margo Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency*, 6(3):34–41, 1998.
- [40] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *19th ACM SOSP*, 2003.
- [41] Tool Interface Standard (TIS) Committee. Executable and linking format (ELF) specification, May 1995.
- [42] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.
- [43] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *1st USENIX Workshop on Offensive Technologies*, August 2007.
- [44] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages

Yan Tang, Qi Gao, and Feng Qin
The Ohio State University
{tangya, gaoq, qin}@cse.ohio-state.edu

Abstract

Continuous memory leaks severely hurt program performance and software availability for garbage-collected programs. This paper presents a *safe* method, called LeakSurvivor, to tolerate continuous memory leaks at runtime for garbage-collected programs. Our main idea is to periodically swap out the “Potentially Leaked” (PL) memory objects identified by leak detectors from the virtual memory to disks. As a result, the virtual memory space occupied by the PL objects can be reclaimed by garbage collectors and available for future uses. If a swapped-out PL object is accessed later, LeakSurvivor will restore it from disks to the memory for correct program execution. Furthermore, LeakSurvivor helps developers to prune false positives.

We have built the prototype of LeakSurvivor on top of Jikes RVM 2.4.2, a high performance Java-in-Java virtual machine developed by IBM. We conduct the experiments with three Java applications including Eclipse, SPECjbb2000 and Jigsaw. Among them, Eclipse and Jigsaw contain memory leaks introduced by their developers, while SPECjbb2000 contain a memory leak injected by us. Our results show that LeakSurvivor effectively tolerates memory leaks for two applications (Eclipse and SPECjbb2000), i.e., no cumulative performance degradation and no software failures when facing continuous memory leaks at runtime. For Jigsaw, LeakSurvivor extends the program lifetime by two times and improves the performance by 46% compared with native runs. Furthermore, when there are no memory leaks, LeakSurvivor imposes small runtime overhead, i.e., 2.5% over the leak detector and 23.7% over the native runs.

1 Introduction

Garbage-collected languages such as Java and C# become increasingly popular. This is partly because programs written in these languages are free of many types of memory errors, which are notorious for compromising system availability and security [42]. For example, by forbidding explicit pointers, Java programs do not encounter memory corruptions due to incorrect pointer

arithmetics. With tremendous advances of hardware and Just-In-Time (JIT) compiler techniques, garbage-collected languages are now applied even in enterprise server environments [3, 7].

Unfortunately, memory leaks still exist and severely affect performance and availability for garbage-collected programs, even though garbage collectors can reclaim unreachable memory objects [23]. Memory leaks occur in a garbage-collected program when the program keeps references to memory objects that are no longer needed. As a result, the heap space occupied by leaked memory objects cannot be reclaimed by garbage collectors. For long-running garbage-collected programs, continuously-leaked memory objects take up more and more space in the heap, leading to more frequent invocation of garbage collections (GC) at runtime. Additionally, more leaked objects in the heap space increase the object traversal time during each GC phase. Therefore, continuous memory leaks cumulatively degrade overall program performance [12, 27]. Even worse, memory leaks may exhaust system resources, eventually causing program crashes [18, 27].

The performance degradation problem due to memory leaks cannot be completely solved by the paging mechanism used in OS memory management or by infinitely increasing the heap size (e.g., in 64-bit machines). The OS paging mechanism swaps out physical memory to disks when physical memory is under pressure, which leaves the virtual memory space un-reclaimed. Therefore, heap pressure (i.e., little available heap space) in the virtual memory due to memory leaks remains the same, leading to cumulative performance degradation. Alternatively, infinitely increasing the heap size, e.g., in 64-bit machines, reduces the heap pressure and thus eliminates the increasing GC frequency. However, once the working set of garbage collectors, i.e., the whole heap, becomes too large to fit into the available physical memory, the program performance will be drastically degraded due to increased memory paging during each GC phase [44].

Therefore, it is imperative to devise mechanisms for tolerating continuous memory leaks at runtime for garbage-collected programs. The mechanisms must *enable programs to maintain relatively stable perfor-*

mance and survive software failures caused by continuous memory leaks.

There are only a few studies [13, 35, 24, 34] on tolerating memory leaks. They are either unsafe or unable to prevent performance degradation and out-of-memory errors. Cyclic memory allocation [34] tolerates memory leaks by restricting each allocation site to a fixed number of live objects. While this method may work for applications that have predictable memory requirements, it is in general unsafe since the live objects could be overwritten. Melt [13] and Plug [35] isolate leaked objects from non-leaked objects and rely on the OS paging for releasing the physical memory occupied by leaked objects. While they are effective in alleviating the physical memory shortage caused by memory leaks, these methods cannot reduce heap usage in the virtual memory and will eventually trigger the out-of-memory errors. Goldstein et al. proposed to dump large unused objects to disks [24]. However, their approach does not handle small leaked objects, which can insidiously cause performance and availability problems. Furthermore, it lacks an automatic mechanism to detect leaked objects.

Our Contributions. In this paper, we propose a *safe* method for tolerating continuous memory leaks for garbage-collected programs at runtime. It can avoid cumulative performance degradation and software failures due to continuous memory leaks. In addition, it helps developers to prune false positives.

Our idea is to periodically move the “Potentially Leaked” (PL) memory objects identified by leak detectors from the virtual memory to disks. As a result, the heap space occupied by PL objects can be reclaimed by garbage collectors and thus are available for future uses. If a swapped-out PL object is accessed later, LeakSurvivor will restore it from disks to the memory for correct program execution.

Based on the above idea, we build a tool called LeakSurvivor in Jikes RVM [7], a high performance Java-in-Java virtual machine developed by IBM, to tolerate memory leaks at runtime. We evaluate LeakSurvivor with three applications, including Eclipse, a popular integrated development environment, SPECjbb2000, a simulator of an order-processing server system, and Jigsaw, a web server. Among them, Eclipse and Jigsaw contain memory leaks introduced by their developers, while SPECjbb2000 contain a memory leak injected by us. Unlike previous approaches, LeakSurvivor possesses the following advantages:

- LeakSurvivor can safely and effectively tolerate memory leaks for garbage-collected program at runtime. Our evaluation shows that it can tolerate leaks for two of the three applications (Eclipse and SPECjbb2000). For Jigsaw, LeakSurvivor extends

its lifetime by two times and improves performance by 46% compared with native runs.

- LeakSurvivor incurs low overhead for programs during normal execution without memory leaks. This is because it only adds one extra check during GC phase. Our evaluation with DaCapo benchmarks shows that LeakSurvivor incurs small runtime overhead, i.e., 2.5% over the leak detector and 23.7% over the native runs, when there are no leaks.
- LeakSurvivor requires no modification in the applications’ source codes. Therefore, it can be easily applied to legacy garbage-collected programs.
- LeakSurvivor provides false positive information on memory leaks, if any, to developers for pruning purposes. Once a PL object is restored to the memory, LeakSurvivor marks it as a false positive and passes this information to the leak detectors.

2 Main Idea of LeakSurvivor

The main idea of LeakSurvivor is to periodically swap out “Potentially Leaked” (PL) objects from the virtual memory to disks so that garbage collectors can reclaim the heap space occupied by these objects. If all leaked memory objects are identified as PL objects and swapped out, the heap pressure due to memory leaks is eliminated. As a result, LeakSurvivor can avoid cumulative program performance degradation and future program crashes due to continuous memory leaks. Upon accesses to some swapped-out PL objects during subsequent program execution, i.e., PL objects are falsely identified, LeakSurvivor swaps them from disks to memory so that the programs can continue execution correctly. In this way, LeakSurvivor guarantees the safety of leak tolerance. Additionally, LeakSurvivor helps developers to prune false positives.

Figure 1 shows the process by which LeakSurvivor tolerates memory leaks at runtime and guarantees the correctness of subsequent program execution. During program execution, LeakSurvivor periodically checks whether there are PL objects marked by the integrated leak detectors. If so, it swaps out the PL objects by copying the object contents from memory to the leak space (See Section 3.2.1), which is mainly on disks. Additionally, LeakSurvivor modifies all the PL objects’ incoming references, i.e., references from other objects to the PL objects, and outgoing references, i.e., references from the PL objects to other objects (See Section 3.2). After this, the virtual memory space occupied by the PL objects can be safely reclaimed by garbage collectors and are available for future uses. Figure 1(a)(b) shows the

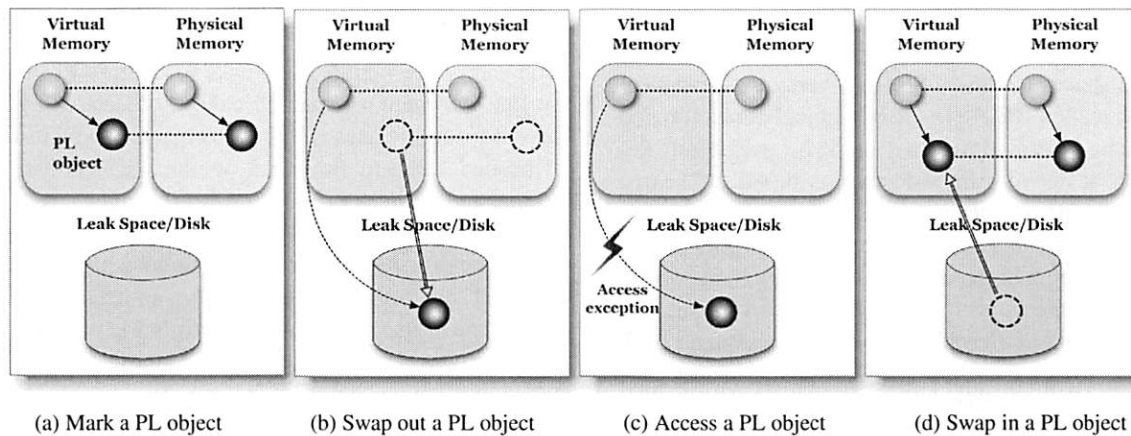


Figure 1: LeakSurvivor: The main idea (PL means “Potentially Leaked”).

swap-out process, which reduces heap pressure due to memory leaks.

Unfortunately, even the state-of-the-art leak detectors report false positives, i.e., PL objects that are not really leaked. Once the program accesses a swapped-out PL object through an incoming reference, which was modified to a unique reserved address during a swapping out phase, an OS exception (segmentation fault) will be triggered. In the exception handling routine, LeakSurvivor identifies the PL object based on the unique reserved address and swaps in the object by copying its content from the leak space to memory and restoring the corresponding incoming and outgoing references. After returning from the exception, the program continues execution correctly by retrying the “faulty” instruction. The swap-in process is shown in Figure 1(c)(d).

3 Design and Implementation

LeakSurvivor consists of three major components that detect and tolerate memory leaks during program execution. As shown in Figure 2, the three components in LeakSurvivor are: (1) leak detectors for detecting “Potentially Leaked” (PL) memory objects; (2) a swap-out component for copying PL objects from the memory to the leak space, which is mainly on disks; (3) a swap-in component for restoring PL objects from the leak space to the memory upon future accesses to them. We implement LeakSurvivor on top of Jikes RVM 2.4.2, a high-performance Java-in-Java virtual machine [7]. We do not see any particular difficulties to implement the ideas of LeakSurvivor in other Java Virtual Machines (JVM).

3.1 Leak Detectors

Leak detectors identify PL memory objects and pass this information to the swap-out component. For garbage-collected programs, memory objects that are unreach-

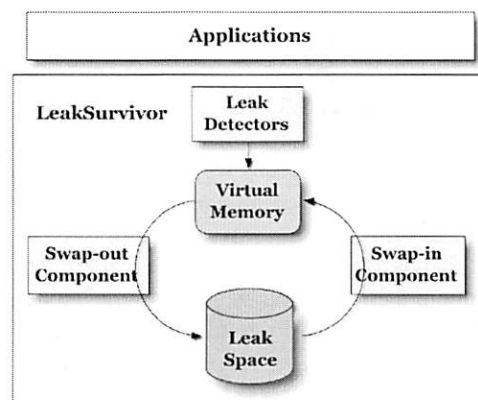


Figure 2: LeakSurvivor architecture

able from *roots*, i.e., references in global, static, and stack variables, are automatically reclaimed by garbage collectors. Therefore, leak detectors for garbage-collected programs only report potentially leaked objects that are still reachable from *roots* as PL objects. There are two types of such leak detectors. The first type is to exploit heap growth and heap difference for identifying leaked objects or data types that cause heap growth. This type of tools include JRockit [43], Leak-Bot [33], and Cork [31], just to name a few. The second type is to exploit the lifetime or staleness of each object for identifying leaked objects that are not used for a long time. In the prototype of LeakSurvivor, we use Sleigh [12], a lightweight, low space overhead leak detector of the second type. The idea of our LeakSurvivor, however, is independent of leak detection methods.

Sleigh tracks accesses to each object at runtime and marks objects that are not accessed for a long time as *stale* objects. More specifically, Sleigh designates two bits as the stale counter for each object to record how long the object has not been accessed. It resets the stale counter to zero upon object allocation and accesses, and increases the stale counter in logarithmic scale at

garbage collection (GC) time. Once both stale bits become ones, the object is deemed as stale. To minimize the space overhead, Sleigh leverages a statistical approach, called BELL, to encode object allocation and last-use site information into a single bit per object. Periodically, it decodes the information based on a large number of stale objects and reports class names, the allocation and last-use sites of stale objects. In its implementation, Sleigh borrows four unused bits in the object header and thus incurs no per-object space overhead. For more details about Sleigh, please refer to the paper [12].

LeakSurvivor considers the stale objects whose class names are reported by Sleigh's decoding phases as PL objects. Sleigh does not report class names for the stale objects with small occurrence since they are most unlikely continuous leaks and much less harmful. Furthermore, LeakSurvivor focuses on application classes instead of primitive types such as char array, classes with the prefix "java.*", etc. Since Sleigh's decoding phase is expensive, we currently run it offline and pass the results to LeakSurvivor. In the future, we plan to extend Sleigh to run the decoding phase online in a separate machine or different cores of the same machine. Additionally, we modified Sleigh to record false positives reported by the swap-in component so that they can assist developers for further bug diagnosis and the falsely identified objects will no longer be marked as PL objects.

3.2 Swap-Out Component

The swap-out component saves the PL objects at runtime so that the virtual memory space occupied by them can be safely reclaimed by garbage collectors and are available for future memory requests. More specifically, after PL objects are identified by leak detectors, the swap-out component copies their content from the virtual memory to the leak space (Section 3.2.1). In addition, the swap-out component assigns all the incoming references to each PL object with a unique reserved address (Section 3.2.2) that is not allowed to be dereferenced at the user level. For all the outgoing references from each PL object, the swap-out component uses a swap-out table (Section 3.2.3) to record such information for facilitating future garbage collections. During subsequent program execution after a swap-out process (Section 3.2.4), if an instruction has nothing to do with any reserved address, it is normally executed. Otherwise, it triggers an OS exception, which will be handled by the swap-in component (Section 3.3) so that the program can correctly continue execution. We summarize the swap-out process in Section 3.2.5.

3.2.1 Leak Space

The leak space mainly uses disks for storing PL objects so that the memory space for these objects can be released for future uses and the PL objects can be restored if needed. Due to slow disk operations, it is essential to manage the leak space for efficient object write and read operations in the leak space. In LeakSurvivor, write operations are dominant since continuous memory leaks make PL objects continuously to be moved to the disks. In the meantime, LeakSurvivor reads PL objects from disks only when they are falsely identified by Sleigh, which has low false positive rates [12].

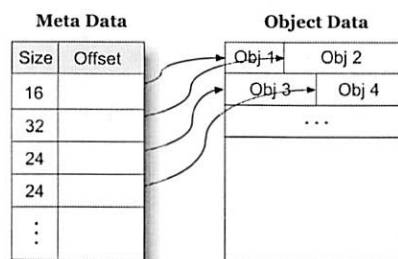


Figure 3: Leak space structure

LeakSurvivor organizes the leak space in a way similar to the Log-structured File System [38] for efficient write accesses of various-sized PL objects and random read accesses. More specifically, a PL object is sequentially appended to the last PL object in the leak space for a write operation. The disk space occupied by the PL objects will not be reclaimed even when the objects are brought back to the memory. This design is to trade disk capacity for slow write operations given the ever-increasing disk capacity per dollar [26]. Once the disk capacity becomes a real problem, we can use compaction techniques [17] to clean up disk space occupied by useless PL objects.

To provide efficient random read accesses to PL objects in the leak space, LeakSurvivor uses fix-sized meta data for each PL object. As shown in Figure 3, the meta data for each PL object consists of its size, its disk address, and other related information specific for Jikes RVM. The index of each entry is the unique id associated with a PL object. With the index number, LeakSurvivor can easily identify the meta data of the PL object and read the object from the leak space.

To further improve the performance of the leak space write and read operations, LeakSurvivor dedicates two chunks of memory as buffers for storing PL objects and their meta data respectively. More specifically, LeakSurvivor first stores a PL object and its meta data in the buffer. Once the buffer is full, the chunk of PL objects and their meta data will be flushed to disks. We currently implement synchronous flushes in LeakSurvivor. For

better performance, we plan to support asynchronous flushes. During our experiments, 8 MB buffer is large enough to provide efficient PL object writes to the leak space. Furthermore, when LeakSurvivor reads a PL object from the leak space, it first checks the buffer to see whether the object is still there. If yes, LeakSurvivor avoids the expensive disk read (See Section 5.2.5).

3.2.2 Incoming References and Reserved Addresses

After a PL object is copied from the memory to the leak space, LeakSurvivor assigns all the incoming references, i.e. references to the PL object, with a unique reserved address to guarantee correct program execution upon future accesses to the PL object. To modify all the incoming references, LeakSurvivor leverages forwarding pointer techniques [17, 21] and integrates the reference modification into the object traversal process at GC phases. More specifically, LeakSurvivor stores the unique reserved address for each PL object in its object header as the forwarding pointer. During the GC phase, when the traversed object has a reference to a PL object, LeakSurvivor modifies its reference to the reserved address stored in the forwarding pointer. As a result, LeakSurvivor avoids one extra live object traversal for modifying incoming references of each PL object.

The reserved addresses are within the address range reserved exclusively for the OS kernel use. Consequently, any access to a PL object via de-referencing the associated reserved address at the user level triggers an OS exception. In the exception handling routine, the swap-in component brings the PL object from the leak space to the memory so that the program can continue execution correctly.

Such use of reserved addresses in LeakSurvivor has no conflict with the use of reserved addresses by the kernel itself. This is because the virtual memory pointed to by reserved addresses can be normally accessed at the kernel level without raising any exception. However, it may cause problems if the reserved addresses for some PL objects are passed from applications to the kernel. To address it, LeakSurvivor intercepts system calls in Java system call wrappers and swaps in the objects before invoking the system calls.

LeakSurvivor maintains a one-to-one mapping between the reserved addresses and the PL objects by sequentially assigning a different reserved address to each new PL object. Equation 1 shows this mapping, where $Addr(obj)$ is the reserved address of a PL object, $Index(obj)$ is the meta data index of a PL object, and $OFFSET$ is the base address of the reserved address range. Therefore, given a reserved address, LeakSurvivor can easily locate the corresponding PL object by calculating the meta data index, and vice versa.

$$Addr(obj) = Index(obj) + OFFSET \quad (1)$$

The reserved address range is large enough for tolerating many leaked memory objects. In a 32-bit machine, OSes such as Linux usually reserve 1GB address range exclusively for the kernel use, which means LeakSurvivor can move around 1G PL objects to the leak space. According to previous study [10], the size of most objects in Java is from 40 to 80 bytes, which includes both the object content and header information. Therefore, LeakSurvivor can tolerate continuous memory leaks with total size of 40-80 GB, which is 10-20 times of the entire virtual memory space, in a 32-bit machine. With emerging 64-bit machines, LeakSurvivor can leverage much more reserved addresses for tolerating memory leaks.

It is obvious that garbage collectors should not follow reserved addresses. Otherwise, it will trigger unnecessary swapping of PL objects from the leak space to the memory. We modify the garbage collector so that it checks whether the currently-being-traversed reference is within the reserved address range or not. If yes, the garbage collector does not de-reference it and continue with other references and objects. Otherwise, the garbage collector executes as usual.

3.2.3 Outgoing References and Swap-Out Table

After a PL object is moved from the memory to the leak space, LeakSurvivor must properly handle its outgoing references for two reasons. First, the outgoing references need to be updated if the pointed-to objects are moved to different memory locations, which can happen for copying garbage collectors [17, 21]. Second, garbage collectors need to traverse the outgoing references during GC phases. Otherwise, objects that are pointed to only by these references will be incorrectly reclaimed.

LeakSurvivor uses a in-memory Swap-Out Table (SOT) to record all the outgoing references for each PL object. More specifically, each SOT entry represents one outgoing reference from a PL object. We modify the garbage collector to add all the outgoing references in the SOT to the *roots* before it starts the object traversal process. Consequently, the garbage collector can efficiently and correctly reclaim unreachable objects without reading PL objects from disks.

As shown in Figure 4, each SOT entry consists of two fields: one is the outgoing reference from the PL object to another object in the memory and the other is the reference counter for recording the number of outgoing references sharing the entry. For handling each outgoing reference from a PL object in the swap-out process, LeakSurvivor first looks it up in the SOT. If any matching entry is found, LeakSurvivor simply increments the reference counter of that entry by one. Otherwise, a new

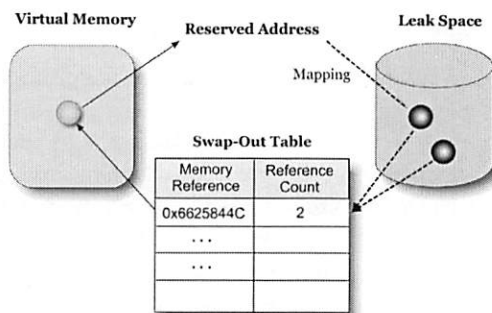


Figure 4: Swap-out table and outgoing references

entry is created for the outgoing reference. After this, LeakSurvivor modifies the field in the PL object from the outgoing reference to the index of the SOT entry.

Since the SOT resides in memory, we must limit its memory consumption from overshadowing the benefits of swapping out PL objects. Fortunately, the memory consumed by the SOT is much smaller than the memory saved by moving PL objects to the leak space. This is because many objects in garbage collected languages, such as Java, only have 1 to 2 incoming reference [10]. In this situation, The children objects of a PL object are also likely leaked since they cannot be reached without accessing the only parent PL object first. Therefore, LeakSurvivor does not need to create a SOT entry for many PL objects upon moving them to the leak space. Additionally, LeakSurvivor reduces the SOT size by reference counters if multiple PL objects have outgoing references to the same object in the memory. Furthermore, LeakSurvivor reclaims a SOT entry if the reference counter becomes zero, which occurs when all the related PL objects are moved back to the memory. Another chance to reclaim a SOT entry is before flushing PL objects from the memory to disks. During this period, LeakSurvivor checks the SOT entries used by the PL objects in the buffers to see whether their outgoing references are modified to some reserved addresses. If any, LeakSurvivor reclaims the SOT entry and modifies the field of the PL object to the reserved address. Our experiments show that a SOT with 1 MB is sufficient to tolerate continuous memory leaks for the evaluated applications (See Section 5.2.4).

3.2.4 Instruction Execution

LeakSurvivor guarantees correct program execution after swapping out PL objects into the leak space. We can verify it by examining all the machine instructions that are possibly executed since Jikes RVM compiles class methods into native code for better performance.

We classify all the machine instructions into three types based on how the instructions are related to the references (i.e., reserved addresses) to PL objects. The first

and simplest type is instructions whose operands have nothing to do with the references to any PL object, i.e., not performing any operations over the references. This type of instructions can be correctly executed without being affected by LeakSurvivor.

The second type of instructions perform non-de-referencing operations such as the equality operation “==” and the assignment “=” over the references to some PL objects. This type of instructions can be correctly executed since LeakSurvivor associates each PL object with a unique reserved address so that the program distinguishes references to different PL objects from each other. For example, after a PL object A is moved to the leak space, its incoming reference value will be changed from A_{old} to some reserved address A_{resv} . In this scenario, if A_{old} equals to B_{old} , i.e., both references pointing to the same PL object, then LeakSurvivor ensures that A_{resv} equals to B_{resv} as described in Section 3.2.2, and vice versa. Note, type checking systems in garbage-collected programs prohibit arithmetic operations such as + and - over references [25].

The last type of instructions de-reference the references to some PL objects, which triggers OS exceptions due to the reserved addresses. Such instructions are compiled from operators such as field access, method invocation, *instanceof*, *type cast*, etc. [25]. They attempt to access either the content or the header information of the PL objects. Therefore, LeakSurvivor needs to bring the corresponding PL objects to the memory for continuing program execution correctly. Array bounds checking, for example, will trigger the swapping in of the array object itself, but leave the objects referenced by array elements untouched.

3.2.5 Swap-out Process

There are three steps for moving a PL object to the leak space and modifying the outgoing references. First, LeakSurvivor attempts to move the children objects to the leak space if they are stale. Otherwise, LeakSurvivor skips them since they are being actively accessed. Second, the content of the PL object is copied from the memory to the leak space. We use the depth-first order to avoid unnecessary SOT entries if children objects are also swapped out. The last step is to create a SOT entry for each outgoing reference that is still pointing to an object in memory.

LeakSurvivor integrates the swap-out process with GC's object traversal process to achieve better performance. More specifically, when each object is being traversed by the garbage collector, LeakSurvivor checks whether it is stale and its class name reported by Sleigh. If yes, it is a PL object and LeakSurvivor swaps it out via the above three steps. Additionally, if a stale object

is pointed to by the reference of any SOT entry, LeakSurvivor swaps it out since it is likely to be a PL object due to its staleness and its ancient PL object. After an object is swapped out, the garbage collector will assign all the future incoming references to this object with its associated reserved address using forwarding pointer techniques. Note, we modify the garbage collector to add the references in all the SOT entries to the *roots* before the object traversal.

When there are circular references among some swapped-out objects, the above swap-out process creates one SOT entry recording the outgoing reference from the last PL object to the first PL object. This is incorrect because after the swap-out process the first PL object is moved to the leak space, which makes this outgoing reference in the SOT entry obsolete. To address it, LeakSurvivor inspects all the SOT entries to see whether the outgoing references are pointing to some objects in the leak space. If so, LeakSurvivor modifies the fields in the corresponding PL objects to the reserved addresses.

3.3 Swap-in Component

To handle an OS exception due to de-referencing a reserved address, the swap-in component first identifies the target PL object (Section 3.3.1), allocates virtual memory space for the PL object, copies its content from leak space to the memory, and restores its incoming and outgoing references (Section 3.3.2). Additionally, the swap-in component notifies the leak detectors of the falsely-identified PL object. After this, the program automatically retries the “faulty” instruction and continues the execution correctly. We discuss the multi-threading issue in Section 3.3.3 and summarize the swap-in process in Section 3.3.4.

3.3.1 PL Object Identification

In an OS exception, LeakSurvivor checks whether it is a segmentation fault. If yes, LeakSurvivor retrieves the de-referenced reserved address from the base register in the faulty instruction. Otherwise, it passes the exception to the default handlers. Given a reserved address $Addr(obj)$, LeakSurvivor can calculate the meta data index of the PL object, i.e., $Addr(obj) - OFFSET$, based on Equation 1. Then, LeakSurvivor fetches the information of the PL object and reads it from the leak space.

The above scheme works in most cases since the compiler often generates code with $base_address + offset$ addressing mode, where $base_address$ is the starting address of an PL object. However, the compiler occasionally generates code that first calculates the result of $base_address + offset$, then stores the result in the base register, and finally de-references the address stored in

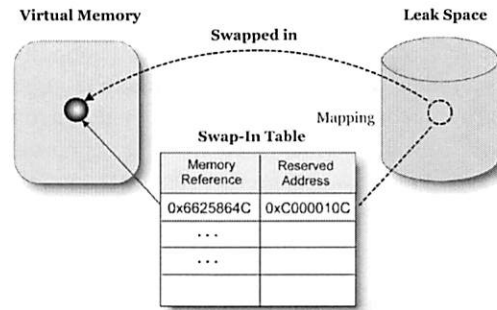


Figure 5: Swap-in table and incoming references

the base register. In this case, LeakSurvivor incorrectly considers the value of $base_address + offset$ stored in the base register as another PL object. To solve this problem, we currently modify the compiler to only generate code with $base_address + offset$ addressing mode. We plan to extend LeakSurvivor to solve this problem more elegantly based on ideas of derived pointers [20].

3.3.2 Incoming and Outgoing References

After copying the PL object from the leak space to the memory at a new location, LeakSurvivor needs to restore its incoming and outgoing references. The incoming references come from three sources: registers, the memory, and the leak space. For incoming references from registers, LeakSurvivor scans the general registers and modifies their values to the new memory address if they are the reserved address.

For incoming references from the memory, one simple way is to traverse all the live objects from *roots* and modify the incoming references from the reserved address to the new memory address of a swapped-in PL object. For better performance, LeakSurvivor delays this modification process to the next GC phase with the help of the swap-in table described as below.

For incoming references from other PL objects in the leak space, it is prohibitively expensive to scan all the PL objects and make modifications. LeakSurvivor introduces a Swap-In Table (SIT) to address this issue. As shown in Figure 5, each SIT entry records the mapping from the reserved address of a PL object to its new memory address after being swapped in. With the SIT, LeakSurvivor can leave the incoming references from the leak space untouched until the PL object is swapped back to the memory. In such situation, LeakSurvivor looks up the incoming reference in the SIT and modifies the reference to the new memory address.

In summary, LeakSurvivor only modifies the incoming references from registers at the time of handling the exception and retries the “faulty” instruction. This method may severely hurt overall program performance in some scenarios, although we have not experienced

such situation in the experiments. For example, if a program de-references a reserved address in a loop, it may raise exceptions repetitively by the same faulty instruction since LeakSurvivor does not modify the incoming reference from the memory. To alleviate this problem, we can modify local variables that contain incoming references to the new memory address by scanning the stack frames. Also we can modify the incoming references in the memory by performing live object traversal to avoid such repetitive exceptions.

The SIT is small due to the low false positive rate of the leak detector [12]. Our experimental results in Section 5 show that 0.5 MB SIT size is big enough for all the three evaluated applications. If the SIT becomes too large, it indicates that the false positive of the leak detector is high and we should try other detectors.

For the outgoing references in the PL object that is being swapped into the memory, LeakSurvivor resolves them by looking up the SOT. More specifically, for each outgoing reference, LeakSurvivor reads the corresponding SOT entry, and modifies the outgoing reference to the reference in that entry. A more sophisticated alternative is to first check whether that SOT entry is pointing to the memory. If so, it is up-to-date and LeakSurvivor will update outgoing reference with it; otherwise, it will search SIT table until either there is no matching result or the reference in the matching entry points to an object in the memory. We implement the lazy method (the first one) because of its simplicity.

3.3.3 Multi-Threading

When an application has multiple threads, more than one thread can access some PL objects by de-referencing the reserved addresses concurrently. This triggers multiple OS exceptions at the same time. Therefore, LeakSurvivor has to be thread-safe for the swap-in process. Similarly, the swap-out process also needs to be thread-safe. The prototype of LeakSurvivor serializes the swap-in and swap-out processes in multiple threads by disabling thread switching once it enters any of the processes. It is equivalent to use a big lock to synchronize the entire swap-in and swap-out processes among multiple threads. For better performance, we plan to extend LeakSurvivor with fine-grained locks to enable concurrent execution of the swap-in and swap-out processes in multiple threads.

3.3.4 Swap-in Process

During the swap-in process, LeakSurvivor first identifies the reserved address of the accessed PL object based on the faulty instruction. Then it looks up the reserved address in the SIT to see whether the PL object has already been swapped into the memory. If hit, LeakSurvivor

modifies the registers that contain the reserved address to the new memory address found in the SIT. Otherwise, LeakSurvivor reads the PL object from the leak space to the memory and inserts the mapping from the reserved address to the new memory address into the SIT. After this, LeakSurvivor updates the PL object's outgoing references based on the result of SOT look-up. Finally, it notifies the leak detector of the false positive.

4 Evaluation Methodology

Our experiments are conducted on two machines, each has a 2.8 GHz Intel Xeon processor, 512 KB L2 cache, 1 GB of memory, and 120 GB hard drive, running with Linux 2.4.27-no-SMP. They are connected by a 100 Mbps Ethernet connection. We run the evaluated applications on one machine. For server programs, we run their clients on the other machine.

We implement LeakSurvivor on top of Jikes RVM 2.4.2, a high performance Java-in-Java virtual machine developed by IBM [7]. The leak detector deployed in LeakSurvivor is Sleigh, a space-efficient, low false positive leak detection tool for Java applications [12]. LeakSurvivor uses a mark-sweep garbage collector because currently Sleigh does not support moving objects. However, we do not see any difficulty in porting LeakSurvivor to copying garbage collectors. For all the experiments, we use the fast adaptive optimizing compiler, and default heap size (100 MB) in Jikes RVM. Additionally, we set the base of the stale counter to 4, the default value in Sleigh.

Application	Version	#LOC	Description
Eclipse	3.1.2	2813358	an integrated development environment
SPECjbb-2000	1.02	30486	a server simulator
Jigsaw	2.0.2	121776	a web server
SPECjbb-2000-fp	1.02	30586	SPECjbb2000 with controlled false positives
DaCapo	2006-10 MR2	N/A	A standard Java benchmark suite

Table 1: Applications used in evaluation

We evaluate LeakSurvivor with four different applications shown in Table 1, including a popular integrated development environment (Eclipse [2]), a simulator of an order-processing server (SPECjbb2000 [6]), a web server (Jigsaw [4]), and a standard Java benchmark suite consisting of 11 benchmarks (DaCapo [10]). Among them, Eclipse and Jigsaw contain memory leaks originally introduced by their developers, while SPECjbb2000 contain a memory leak injected by us. In addition, to evaluate the performance of LeakSurvivor under different false positive rates, we modify

SPECjbb2000 by accessing leaked objects with different controlled rates and rename it as SPECjbb2000-fp. We use DaCapo benchmarks to measure the performance overhead of LeakSurvivor when there is no leaks.

In this paper, we design three sets of experiments to evaluate the key aspects of LeakSurvivor:

- The first set evaluates the functionality of LeakSurvivor in tolerating memory leaks at runtime. In this set of experiments, we run the applications with memory leaks being triggered as fast as possible in two different configurations: one with LeakSurvivor, i.e., running programs in Jikes RVM with Sleigh and LeakSurvivor, and the other without LeakSurvivor, i.e., natively running programs in JikesRVM without Sleigh and LeakSurvivor. We collect the average response time for client programs (Eclipse) and average throughput for server programs (SPECjbb2000 and Jigsaw) and the detailed statistics, e.g., GC time, memory usage, etc.
- The second set evaluates the performance overhead of LeakSurvivor when there is no memory leaks, which tells us how LeakSurvivor performs during normal program execution. We measure the runtime and space overhead.
- The third set evaluates the sensitivity of LeakSurvivor to different false positive rates of leak detectors. We test SPECjbb2000-fp with LeakSurvivor using different controlled false positive rates.

For Eclipse, each comparison between two directories triggers memory leaks. We adopt the script developed at the University of Texas at Austin [12] to repeatedly compare the source code of two versions of Jikes RVM: 2.4.0 and 2.4.1 (109 of 898 files differ; textual diff is 874 lines). We measure the time for each round of such comparison and consider it as the response time.

For SPECjbb2000, we inject a memory leak by modifying the object access order of a transaction queue from First-in First-out to Last-in First-out. We run it with these leak-triggering transactions in an infinite loop and measure the average transactions per second during last 60 seconds as the throughput.

For Jigsaw, the leak occurs during a client disconnection. We write a client with 10 threads, each sends 20 connect and disconnect requests per second concurrently for triggering the memory leaks. In the meantime, we use ab [1] in the Apache Web server suite as a normal client and measure the throughput of Jigsaw. The client ab creates 10 concurrent connections. For each connection, ab continuously sends out requests to fetch different files whose sizes range in 1 KB, 2 KB, 4 KB, ..., 256 KB with uniform distribution.

5 Experimental Results

5.1 Overall Results

Table 2 shows the overall effectiveness of LeakSurvivor in tolerating memory leaks. For each buggy application, the table shows whether the program crashes and how long it executes before crash, if any, in two different configurations: one with LeakSurvivor and the other without LeakSurvivor. The table also shows the ratio of the program performance with LeakSurvivor to the performance without LeakSurvivor. The performance is averaged over the period when both of the programs are alive. The last three columns in the table show the total sizes of swapped-out objects and live objects, i.e., objects reachable from *roots*, when programs are executing with LeakSurvivor and without LeakSurvivor.

As shown in Table 2, LeakSurvivor successfully tolerates memory leaks for two of the three applications, including Eclipse and SPECjbb2000. Without LeakSurvivor, they crash after 555 seconds' and 328 seconds' execution respectively. This is because continuously-leaked memory objects gradually occupy the heap space and quickly push the total size of live objects to reach their upper bounds, 54.7 MB and 55.3 MB respectively. Note, the live object region is part of the whole heap in Jikes RVM, whose default maximal size, 100 MB, is used in our experiments. In contrast, with LeakSurvivor, the two applications do not crash even after around 2 hours and still maintain relatively stable performance. In this situation, we believe LeakSurvivor successfully tolerate memory leaks and forcefully terminate the execution. LeakSurvivor can tolerate memory leaks because it swaps out leaked memory objects and releases virtual space occupied by them for future uses. Therefore, the total size of the live objects can be kept under a certain percentage of the maximal heap size. For example, LeakSurvivor swaps out 673 MB memory objects and maintains its live object size under 34 MB for Eclipse before we terminate it.

LeakSurvivor does not survive Jigsaw since it has "semantic" memory leaks, which are not detected by the integrated leak detector. The class name of leaked objects in Jigsaw is *HashTableEntry*, which is a *<key, value>* pair. However, the leak detector can only report the *value* part as PL objects. This is because the *key* part is still accessed from time to time for hash table lookup and rehashing. Therefore, LeakSurvivor only swaps out the *value* part of the leaked entry and the *key* part in the heap cumulatively degrades the program performance and eventually crashes the program. Nonetheless, LeakSurvivor still helps extend the program lifetime by two times — the program crashes at 3402 seconds when running with LeakSurvivor and crashes at

Apps	Tolerable?		Live Time (sec)		Performance Ratio* (w/ LS : w/o LS)	Total Size of Objects (MB)		
	w/ LS	w/o LS	w/ LS	w/o LS		SO (w/ LS)	Live (w/ LS)	Live (w/o LS)
Eclipse	Yes	No	> 7626	555	1.27:1	> 673	34	55
SPECjbb2000	Yes	No	> 7135	328	1.24:1	> 857	28	55
Jigsaw	No	No	3402	1147	1.46:1	82	45	52

Table 2: Overall results of LeakSurvivor. w/ LS means with LeakSurvivor, w/o LS means without LeakSurvivor, SO means Swapped-Out. Live objects are objects reachable from roots. *Performance ratio is the ratio of averaged program performance (response time for Eclipse and throughput for SPECjbb2000 and Jigsaw) with LeakSurvivor to that without LeakSurvivor during the period when both are still executing.

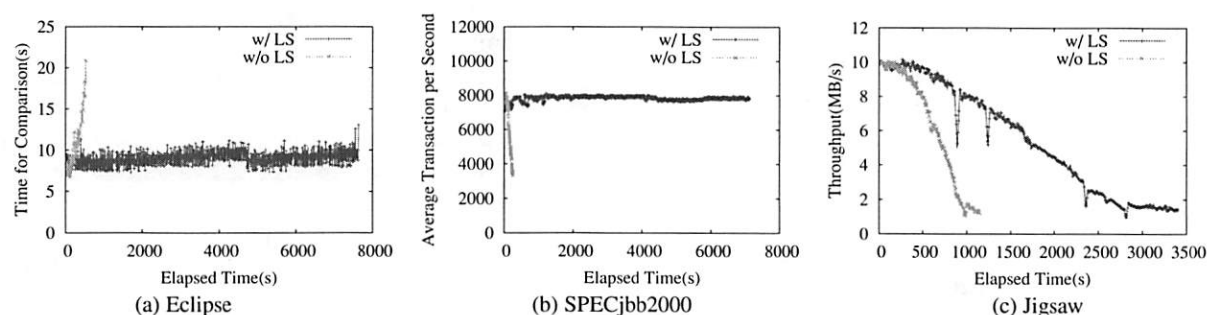


Figure 6: Overall performance for applications w/ and w/o LeakSurvivor

1147 seconds when running without LeakSurvivor. This is because LeakSurvivor swaps out part of the leaked objects, 82 MB out of 120 MB total leaked objects.

Table 2 also shows that LeakSurvivor improves the overall program performance when memory leaks occur continuously. For example, Jigsaw with LeakSurvivor provides 46% more average throughput than that without LeakSurvivor when both are alive. This is mainly because continuously leaked objects occupy more and more heap space, which in turn invokes GC more often, when executing programs without LeakSurvivor. Furthermore, the object traversal time in GC phases also increases due to more live objects. On the contrary, executing programs with LeakSurvivor does not suffer from these two problems because LeakSurvivor keeps swapping out leaked objects and maintains low heap pressure.

5.2 LeakSurvivor Performance with Application Leaks

5.2.1 LeakSurvivor Performance

We measure the performance for the three applications with and without LeakSurvivor when memory leaks occur continuously. As shown in Figure 6, LeakSurvivor helps avoid cumulative performance degradation and software failures for Eclipse and SPECjbb2000. For example, with LeakSurvivor, the comparison time for Eclipse is within the range from 7.2 to 11.3 seconds without increasing trends before we manually terminate the program at around 2 hours. In contrast, without LeakSurvivor, the comparison time for Eclipse drastically increases from 6.7 to 20.9 seconds within around

9.5 minutes before its crash. Although LeakSurvivor cannot fully tolerate leaks in Jigsaw, it alleviates the degree of cumulative performance degradation caused by continuously leaked memory objects. With LeakSurvivor, Jigsaw takes around 2580 seconds for the throughput degraded to below 2 MB per second, while it only takes about 920 seconds for the same performance degradation without LeakSurvivor.

Figure 6 also shows that the performance of programs without LeakSurvivor is slightly better than the performance of programs with LeakSurvivor at the initial period of program execution. For example, Eclipse without LeakSurvivor outperforms Eclipse with LeakSurvivor by 9.7% on average during the execution period from 0 to 200 seconds. This is because, at the initial period of program execution, the runtime overhead incurred by LeakSurvivor and the leak detector is larger than the performance degradation due to continuous memory leaks.

5.2.2 Live Objects Size

Figure 7 shows that LeakSurvivor can effectively control the growth of live object sizes when memory leaks continuously occur, which contributes significantly to the relatively stable overall program performance. For example, without LeakSurvivor the live object size in Eclipse increases sharply from 30 MB to 55 MB before the program crashes. In contrast, with LeakSurvivor the live object size in Eclipse is bounded between 33 MB and 41 MB. This is because LeakSurvivor swaps out PL objects periodically, which in most cases brings down the live object size from 38 to 34 MB when the live object size reaches 38 MB due to continuously-leaked

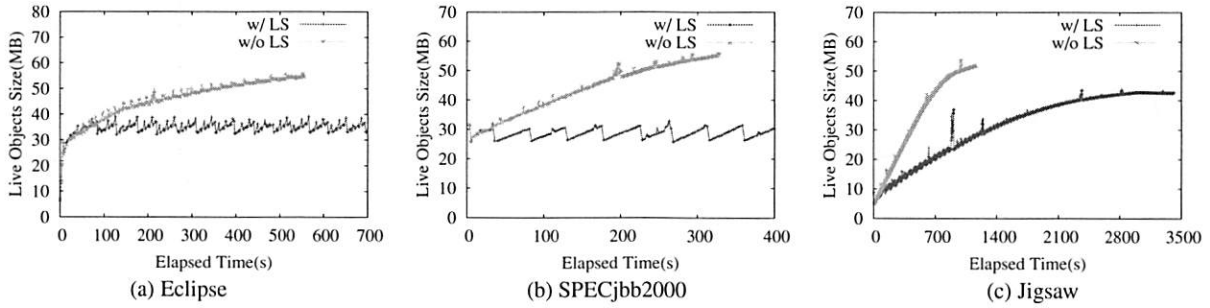


Figure 7: Live object sizes for w/ and w/o LeakSurvivor

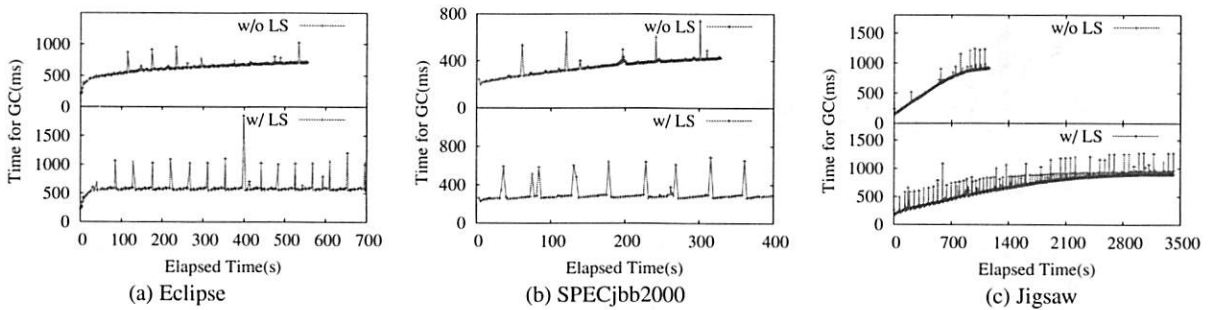


Figure 8: GC performance for w/ and w/o LeakSurvivor

memory objects. When Eclipse is forcefully terminated, LeakSurvivor swaps out 673 MB PL objects, while its live object size is still under 37 MB. For better readability, Figure 7(a) and (b) show shorter time ranges.

5.2.3 GC Performance

Figure 8 shows the time for performing one GC phase, which is directly related to the performance in garbage-collected programs. With LeakSurvivor, the GC time is mostly stable except for some spikes, which are caused by swapping out PL objects at certain GC phases synchronously. However, the overhead of the swap-out process in LeakSurvivor is amortized among all the GC phases and thus not reflected in the overall program performance. For example, the average GC time for Eclipse is 0.607 seconds, while the average non-spike GC time for Eclipse is 0.585 seconds. In other words, the amortized overhead caused by the swap-out process in LeakSurvivor is 3.7 % for GC time.

In contrast, without LeakSurvivor, the GC time steadily increases. For example, the GC time for Eclipse increases from 0.414 to 0.711 seconds right before it crashes. The increased GC time partially contributes to the performance degradation shown in Figure 6.

More importantly, without LeakSurvivor, the GC frequency dramatically increases (not readable in the figure), which severely degrades the program performance. For example, at the beginning of Eclipse's execution, the GC frequency is 23 GCs per minute, and it becomes

Apps	SOT Size (MB)	SIT Size (MB)	LS Buf Size (MB)
Eclipse	0.56	0.02	8
SPECjbb2000	0.66	0.20	8
Jigsaw	0.53	0.00008	8

Table 3: LeakSurvivor space overhead. LS means Leak Space, Buf means Buffer.

51 GCs per minute right before it crashes. In contrast, the GC frequency remains stable for programs with LeakSurvivor. For example, the GC frequency for Eclipse with LeakSurvivor falls within the range of 21 to 30 GCs per minute.

5.2.4 LeakSurvivor Space Overhead

Table 3 shows the space overhead of LeakSurvivor for the three applications before their termination or crash. The space overhead of LeakSurvivor is relatively small (8.02 MB – 8.86 MB). It consists of three parts: SOT, SIT, and the leak space buffer. The space overhead comes predominantly from the leak space buffer, which has fixed size, i.e., 8 MB, for better swap-out performance. The SIT size depends on false positive rates, which are 0.001%, 0.17%, and 0.0004% for Eclipse, SPECjbb2000, and Jigsaw respectively. The SOT size is small and increases very slowly compared with the size of swapped out objects. For example, the SOT size for Eclipse increases from 0.51 MB (0 entries) to 0.56 MB (35270 entries) before it is forcefully terminated at

Apps	SOT Entry #	SwapOut Obj #	SIT Entry #	SIT Hit #	Buf Hit #	Disk Hit #	Exp #
Eclipse	35270	19293061	200	155816	189	11	78163
SPECjbb2000	41419	15020902	25277	59285	18967	6310	46219
Jigsaw	108	2831100	10	10	10	0	20

Table 4: Swap-in & swap-out statistics. Buf means Buffer, Exp means Exception. SIT Hit #, Buffer Hit #, and Disk Hit # are the number of hits when LeakSurvivor looks up reserved addresses in the swap-in table, the leak space buffer, and disks, respectively, during swap-in operations.

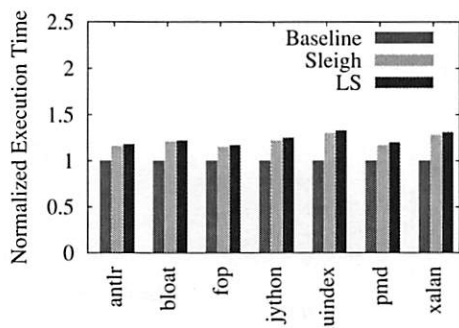


Figure 9: Performance of DaCapo benchmarks. We run the benchmarks in three different configurations: (1) Baseline, i.e., without Sleigh and LeakSurvivor; (2) Sleigh, i.e., only enabling Sleigh; (3) LS, i.e., enabling both Sleigh and LeakSurvivor.

7626 seconds, after swapping out 19293061 PL objects with total size of 673 MB.

There are two reasons for the small SOT size. First, LeakSurvivor swaps out the children PL objects before swapping the parent so that it does not need one SOT entry to record the outgoing reference from the parent PL object. Second, the SOT uses reference counters to share entries if possible.

5.2.5 Exceptions and False Positives

Table 4 shows that SIT and the leak space buffer help reduce the swap-in overhead. First, once an object has been swapped in, all the subsequent exceptions incurred at the same reserved address only need to search SIT and update new references. More than 70% of swap-in operations belong to this category. Second, more than 75% of the first-time swap-in operations hit in the leak space buffer. For this case, we only need a *memcpy* to move the PL object back from the in-memory buffer without disk reads. In addition, Table 4 shows the false positive rates of the leak detector (SIT Entry # / SwapOut Obj #), i.e., 0.001%, 0.17%, and 0.0004% for Eclipse, SPECjbb2000, and Jigsaw respectively, are low.

5.3 LeakSurvivor Overhead without Application Leaks

We measure the runtime overhead incurred by LeakSurvivor when running with programs that have no mem-

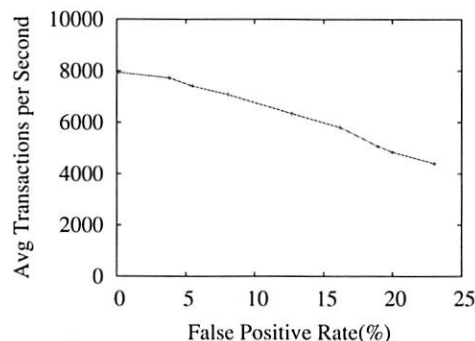


Figure 10: LeakSurvivor performance for SPECjbb2000 with different false positive rates

ory leaks. Figure 9 shows the performance of DaCapo benchmarks in three different configurations: baseline without Sleigh and LeakSurvivor, with Sleigh, and with LeakSurvivor (including Sleigh). The results show that LeakSurvivor incurs small runtime overhead, i.e., 2.5%, on top of Sleigh. The overhead mainly comes from one extra PL object checking for each object traversal during GC phases and slightly increased boot image size (51.30MB for LeakSurvivor v.s. 50.61MB for Sleigh).

Compared with the baseline, LeakSurvivor (including Sleigh) incurs 23.7% runtime overhead, which is mainly caused by the instrumentation code for tracking memory object accesses. Additionally, LeakSurvivor's internal data structure imposes no space overhead when there is no leaks because the SOT, SIT, and the leak space buffer are created on demand.

5.4 Sensitivity Study

We conduct experiments to examine how sensitive the LeakSurvivor is to different false positive rates. As shown in Figure 10, the performance of LeakSurvivor can be severely hurt by the large false positive rates. For example, the average throughput decreases from 7920 to 4400 transactions per second when the false positive rates increase from 0.16% to 23.03%. This is because the overhead incurred by the swap-in process increases dramatically once the false positives increase. Figure 10 also shows that LeakSurvivor's performance is acceptable if the false positive rates are within 5%, which serves as a guide for selecting leak detectors.

6 Related Work

Our work builds on much previous work. Due to the space limit, this section briefly describes the related work that is not discussed in previous sections.

General Fault Tolerance. A considerable amount of research, such as software rejuvenation [30, 22, 11], micro-rebooting [15, 14], and checkpointing-based re-execution methods [8, 45, 16, 32], has been conducted in surviving general software failures. These approaches can deal with failures caused by memory leaks, but programs still suffer cumulative performance degradation before restart and may repetitively fail even after re-execution given that most memory leaks are deterministic. Recently proposed approaches including Rx [37] and DieHard [9] exploit data diversity for tolerating many types of deterministic bugs such as buffer overflows, dangling pointers, and double free, but they do not address memory leaks.

Memory Leak Detection. Memory leaks can be detected statically or dynamically. While static methods [28] can detect some memory leaks without incurring runtime overhead, they may report many false positives due to lack of runtime information. To dynamically detect memory leaks, Purify [27] and Valgrind [39] track object references to identify unreachable objects for C/C++ programs. For garbage-collected programs, unreachable objects are implicitly reclaimed by GCs [23] and thereby only useless objects threaten system availability. JRockit [43], .NET Memory Profiler [40], JProbe [5], LeakBot [33], and Cork [31], track heap updates to identify objects that cause the heap to grow. Other methods, such as SWAT [18], SafeMem [36], and Sleight [12], leverage object lifetime or staleness (time since last use) to identify leaked objects.

Other Related Work. Bookmarking collection [29] can be used to save physical memory from memory leaks at page granularity although its primary goal is to reduce garbage collection overhead. It has the same problem as Melt [13] because it fails to reclaim the heap space.

LeakSurvivor reclaims the PL objects and thereby the space can be reused by the application, which may result in “hot” objects clustered together. This may improve program performance as done in various prior work on data layout optimizations [19]. The object recovery mechanism exploited by the Swap-In component is related to OS page faults [41]. The SIT is related to forwarding pointer techniques used by Copying GC [17, 21].

7 Conclusions and Future Work

In summary, LeakSurvivor is a safe and non-invasive method to tolerate continuous memory leaks at runtime for garbage-collected programs. It helps programs to avoid cumulative performance degradation and software failures due to continuous memory leaks. It does so by swapping out potentially-leaked memory objects to disks and reclaiming virtual memory space occupied by them. LeakSurvivor is safe because it swaps in the swapped-out objects to the memory upon future accesses to them if they are falsely identified as leaks. Additionally, LeakSurvivor assists developers to diagnose memory leaks by providing false positives information due to swapped-in objects. Furthermore, it requires no modification to applications’ source code.

We evaluate LeakSurvivor with three applications that contain continuous memory leaks. The results show that LeakSurvivor can effectively tolerate memory leaks for two of the applications (Eclipse and JBB2000) and extend the lifetime of one application (Jigsaw) by two times. Without LeakSurvivor, all the three applications severely suffer cumulative performance degradation and eventually crash within 20 minutes. This indicates that safely reclaiming virtual memory space occupied by potentially-leaked objects is an effective way to tolerate memory leaks. In addition, LeakSurvivor improves the performance of programs with continuous memory leaks by 24%–46%.

We plan to extend our work in several dimensions in the future. First, we will evaluate LeakSurvivor using more applications with memory leaks. We currently only have three applications since it is difficult to find real-world applications with well-documented memory leaks. Second, we plan to support LeakSurvivor with asynchronous disk flushes and derived pointers. Third, we will investigate the idea of LeakSurvivor for tolerating memory leaks in C/C++ programs, which is difficult since there is no type information at the binary level.

8 Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback on this paper. We appreciate that Yu Chen and Ming Wu in Microsoft Research Asia provided insightful comments at the very early stage of this project. We are indebted to Michael Bond and Dr. Kathryn McKinley in generous sharing and answering questions about Sleight and the script in addition to their invaluable comments. We thank Enhua Tan for setting up the experiment platform, as well as Shuang Liang, Feng Chen and Xiaoning Ding for discussion. We appreciate enormous support for this project from Dr. Xiaodong Zhang.

References

- [1] ab - Apache HTTP Server Benchmarking Tool. <http://httpd.apache.org/docs/2.0/programs/ab.html>.
- [2] Eclipse - An Open Development Platform. <http://www.eclipse.org>.
- [3] Java 2 Platform, Enterprise Edition. http://java.sun.com/j2ee/reference/whitepapers/j2ee_guide.pdf.
- [4] Jigsaw - W3C's Server. <http://www.w3.org/Jigsaw/>.
- [5] JProbe. <http://www.javaperformancetuning.com/tools/jprobe/>.
- [6] SPECjbb2000, A Java Business Benchmark. <http://www.spec.org/osg/jbb2000>.
- [7] B. Alpern, S. Augart, S. M. Blackburn, and et al. The Jikes Research Virtual Machine Project: Building An Open-source Research Community. *IBM Syst. J.*, 44(2):399–417, 2005.
- [8] C. Amza, Armando Cox, and W. Zwaenepoel. Data Replication Strategies for Fault Tolerance and Availability on Commodity Clusters. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, Jun 2000.
- [9] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM conf. on Programming language design and implementation(PLDI'06)*, 2006.
- [10] S. M. Blackburn, R. Garner, and et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM conf. on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*, New York, NY, USA, October 2006.
- [11] Andrea Bobbio and Matteo Sereno. Fine Grained Software Rejuvenation Models. In *International Computer Performance and Dependability Symposium*, Sep 1998.
- [12] Michael D. Bond and Kathryn S. McKinley. Bell: Bit-encoding Online Memory Leak Detection. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, Oct 2006.
- [13] Michael D. Bond and Kathryn S. McKinley. Tolerating Memory Leaks. Technical report, UT Austin Technical Report TR-07-64, Dec 2007.
- [14] George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, and Rakesh Gowda. Reducing Recovery Time in A Small Recursively Restartable System. In *Intl. Conf. on Dependable Systems and Networks*, Jun 2002.
- [15] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, Dec 2004.
- [16] Y. Chen, James S. Plank, and Kai Li. CLIP: A Checkpointing Tool for Message-passing Parallel Programs. In *ACM/IEEE Supercomputing Conference (SC'97)*, Nov 1997.
- [17] C. J. Cheney. A Non-recursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677, November 1970.
- [18] Trishul M. Chilimbi and Matthias Hauswirth. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, Oct 2004.
- [19] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious Structure Layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.
- [20] A. Diwan, E. Moss, and R. Hudson. Compiler Support for Garbage Collection in A Statically Typed Language. In *ACM Conf. on Programming Language Design and Implementation (PLDI '92)*, June 1992.
- [21] Robert R. Fenichel and Jerome C. Yochelson. A Lisp Garbage Collector for Virtual Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, November 1969.
- [22] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. On The Analysis of Software Rejuvenation Policies. In *Proceedings of the Annual Conference on Computer Assurance*, Jun 1997.
- [23] Richard Gillam. An Introduction to Garbage Collection. http://oss.software.ibm.com/icu/docs/papers/cpp_report/an_introduction_to_garbage_collection_part_i.html.
- [24] Maayan Goldstein, Onn Shehory, and Yaron Weinsberg. Can Self-healing Software Cope with Loitering? In *SOQUA '07: Fourth international workshop on Software quality assurance*, pages 1–8, New York, NY, USA, 2007. ACM.
- [25] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
- [26] E. Grochowski and R. D. Halem. Technological Impact of Magnetic Hard Disk Drives on Storage Systems. *IBM Syst. J.*, 42(2):338–346, 2003.
- [27] R. Hasting and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the USENIX Winter 1992 Technical Conference*, Dec 1992.
- [28] David L. Heine and Monica S. Lam. A Practical Flow-sensitive and Context-sensitive C And C++ Memory Leak Detector. In *ACM conf. on Programming language design and implementation (PLDI'03)*, 2003.
- [29] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage Collection without Paging. *SIGPLAN Not.*, 40(6):143–153, 2005.
- [30] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, Jun 1995.
- [31] Maria Jump and Kathryn S. McKinley. Cork: Dynamic Memory Leak Detection for Java. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan 2007.
- [32] David E. Lowell and Peter M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. Technical report, CSE-TR-410-99, Univ. of Michigan, 1998.
- [33] Nick Mitchell and Gary Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *European Conference on Object-Oriented Programming, (ECOOP '03)*, 2003.
- [34] Huu Hai Nguyen and Martin Rinard. Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 15–30, New York, NY, USA, 2007. ACM.
- [35] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Plug: Automatically Tolerating Memory Leaks in C and C++ Applications. Technical report, UMass CS Technical Report 08-09, April 2008.
- [36] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Intl. Symposium on High-Performance Computer Architecture*, Feb 2005.
- [37] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failure. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, Oct 2005.
- [38] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of A Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.
- [39] J. Seward. Valgrind, An Open-source Memory Debugger for x86-GNU/Linux. available at URL <http://developer.kde.org/sewardj/>.
- [40] SciTech Software. .NET Memory Profiler. <http://www.scitech.sel-memprofiler/>.
- [41] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
- [42] US-CERT. US-CERT Vulnerability Notes Database. <http://www.kb.cert.org/vuls>.
- [43] BEA WebLogic. JRockit: Java for The Enterprise. http://www.bea.com/content/news_events/white_papers/BEA_Rockit_wp.pdf.
- [44] Ting Yang, Emery D. Berger, Scott F. Kaplan, J. Eliot B. Moss, and B. Moss. CRAMM: virtual Memory Support for Garbage-collected Applications. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.
- [45] Yuanyuan Zhou, Peter M. Chen, and Kai Li. Fast Cluster Failover Using Virtual Memory-Mapped Communication. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Jun 1999.

Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing

Dan Wendlandt David G. Andersen Adrian Perrig
Carnegie Mellon University

Abstract

The popularity of “Trust-on-first-use” (Tofu) authentication, used by SSH and HTTPS with self-signed certificates, demonstrates significant demand for host authentication that is low-cost and simple to deploy. While Tofu-based applications are a clear improvement over completely insecure protocols, they can leave users vulnerable to even simple network attacks. Our system, PERSPECTIVES, thwarts many of these attacks by using a collection of “notary” hosts that observe a server’s public key via multiple network vantage points (detecting localized attacks) and keeps a record of the server’s key over time (recognizing short-lived attacks). Clients can download these records on-demand and compare them against an unauthenticated key, detecting many common attacks. PERSPECTIVES explores a promising part of the host authentication design space: Trust-on-first-use applications gain significant attack robustness without sacrificing their ease-of-use. We also analyze the security provided by PERSPECTIVES and describe our experience building and deploying a publicly available implementation.

1 Introduction

Despite decades of research into techniques for establishing secure communication channels for networked applications, many of today’s popular protocols remain vulnerable to *Man-in-the-Middle (MitM)* attacks. Some applications provide no security whatsoever (e.g., HTTP), and others rely on self-signed keys or Diffie-Hellman-like key exchange that can protect against eavesdroppers, but not against active adversaries who can interpose on communication between the two parties.

While MitM attacks are not new, widespread use of shared wireless networks coupled with recent discoveries of *automated* MitM attacks in the wild indicate that the threat is increasingly relevant. For example, the Arpiframe worm uses ARP poisoning to interpose on the HTTP traffic of other hosts on the same LAN [26], while worms exploiting simple vulnerabilities in home routers exposed end-hosts to “drive-by pharming” attacks that use DNS to redirect clients fake versions of security-sensitive websites [9].

Furthermore, a study by Reis et al. used client-side measurements to confirm that real-time snooping and modification of web traffic is a reality in today’s networks [20].

In this paper, we examine a novel approach to authenticating a server’s public key. Traditional approaches to server key authentication, such as a public-key infrastructure (PKI) [7, 5], rely on trusted entities (e.g., Verisign) that grant certificates based on the validation of real-world identities. When done securely, such verification requires significant (often manual) effort. While some network hosts, primarily commercial websites, can afford to pay the high verification cost for these certificates, clients have no simple and effective means to authenticate connectivity to most other hosts on the Internet.

Because the high cost of creating and managing a host PKI presented a substantial barrier to the replacement of completely insecure protocols such as telnet, the *SSH model* of host authentication emerged as a pragmatic solution. Authentication in the SSH model relies on the user’s discretion to decide if an unauthenticated key is valid. Keys deemed valid by the client are cached locally and used to authenticate subsequent communication with the same server. While some users may verify all new or changed server public keys in a secure manner (e.g., by memorizing a key fingerprint or verifying the key via an alternate trusted channel), users often simply *assume the absence of an adversary on the initial connection and accept the initial key without verification*. We refer to this common approach as **Trust-on-first-use (Tofu)** authentication (it is also known as “leap-of-faith” authentication).

The Tofu approach has two primary weaknesses:

1. By accepting any key on the initial connection, users render themselves vulnerable to attack by any adversary either on the path between the user and the server or on a shared wireless LAN.
2. On subsequent connections, the user must still manually determine the validity of any key that conflicts with a cached key. A user who assumes such key changes are valid without verification receives no protection against MitM attacks.

These weaknesses in the Trust-on-first-use approach are particularly severe in the case of websites using self-signed SSL certificates, because web clients tend to visit a large

number of sites, increasing the number of vulnerable initial connections. Moreover, web users often lack the means and/or expertise to manually verify keys.

PERSPECTIVES improves on basic Tofu authentication by having a collection of semi-trusted hosts called *network notaries* periodically probe a large number of network services (e.g., SSH and HTTPS servers) to build a record of the public keys used by those services over time. When a client receives an unauthenticated public key from a service, it contacts the notaries to download the history of keys used by that service. This additional data from diverse network vantage points over a span of time gives clients the “perspective” to make a strictly better security decision: clients can often detect attacks during an initial connection or a key cache conflict, the two scenarios when the standard Tofu authentication is most vulnerable.

Because notaries generate their data using automated network probes, *applications using PERSPECTIVES enjoy the same simplified deployment model as SSH: no certificate authority is needed to verify the identity of server owners and grant them certificates*. Instead, the validity of a service’s key is determined by its existence on the network over time. While the notary infrastructure adds some complexity to a Tofu-based application, it exists independent of both clients and servers. Servers can remain unmodified while updated clients benefit from notary data.

While this paper focuses on protocols that use unauthenticated keys (i.e., the SSH model), PERSPECTIVES can also help even when PKI-signed certificates are used. As we discuss in Section 8, because users often ignore browser security warnings [21, 10], a MitM attacker can fool a user by injecting a bogus self-signed certificate in the place of a PKI-signed certificate. PERSPECTIVES clients can easily detect this attack by comparing the received certificate with those seen by the notaries.

This paper makes four primary contributions:

1. It presents the design of a modular network notary infrastructure that tolerates internal failures and compromises (Sections 3 and 5).
2. It describes a framework for (possibly automated) client policies that aggregate notary replies and determine if a key is trustworthy (Section 4).
3. It analyzes PERSPECTIVES’s ability to resist a variety of network attacks within a realistic threat model, demonstrating that it can protect Tofu-based applications from many MitM attacks (Section 6).
4. It describes the implementation and benchmarking of a publicly available release of PERSPECTIVES, including a robust notary server and modified OpenSSH and Mozilla Firefox clients capable of implementing basic key trust policies (Section 7).

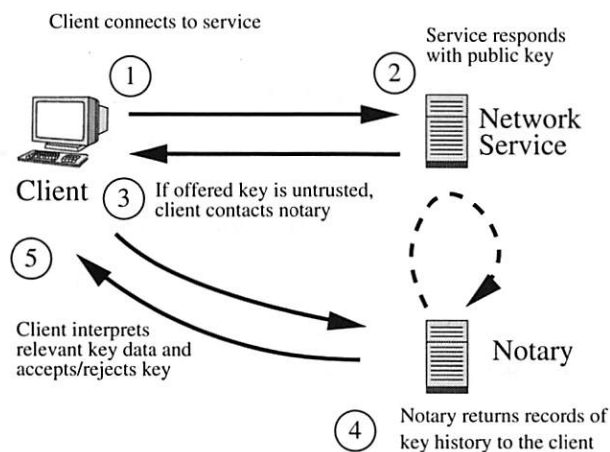


Figure 1: Overview of a client using PERSPECTIVES. In practice, several notaries would be contacted in parallel before making a key trust decision.

2 Overview of PERSPECTIVES

We name our system PERSPECTIVES because it helps clients make sound security decisions by leveraging views from multiple network vantage points. PERSPECTIVES’ task is to help clients determine whether they should accept an untrusted public key received while connecting to a particular *network service*. Example services include SSH access to an end-host or HTTPS access to a website that uses a self-signed SSL certificate.

PERSPECTIVES uses a set of publicly available servers, called *network notaries*, that monitor and record the history of public keys used by a network service. A notary cryptographically signs (i.e., notarizes) statements saying that at time t it observed service S using public key K . The basic operation of PERSPECTIVES is shown in Figure 1. When a client connects to a network service, it receives an *offered key* in reply. If the offered key is unauthenticated (i.e., it does not match an existing key in the client cache) the client must either accept the offered key, taking a security risk, or reject the key, losing the ability to communicate with the service. To obtain more information to make this decision, the client contacts a set of notaries and requests all *observed key data* for that service. The client then uses application-specific *key-trust policies* (Section 4) to interpret this data and accept or reject the key. These policies check for consistency between the offered key and the keys seen by each notary, often allowing clients to distinguish between a legitimate key and an attack.

2.1 Threat Model & Attack Resistance

Attackers mount MitM attacks by providing clients with a false public key in order to observe or modify network communications. In our attack model, an adversary can compromise any path in the network as well as components

of the notary infrastructure itself. Only the client and server themselves must be completely trusted, a standard requirement for host authentication schemes.

While our model allows any network or notary component to be compromised, we borrow from Abraham Lincoln and assume that an attacker “can fool all of the [components] some of the time, and some of the [components] all of the time, but it cannot fool all of the [components] all of the time.” That is, we assume that attacks are *either*: (1) localized to a particular network scope or (2) of limited duration, since a larger attack is more easily detected and remedied.

In this paper, we use the term *redundancy* to describe the protections that PERSPECTIVES provides. Key observations gathered from multiple network vantage points provide *spatial redundancy*, since unless an attacker can compromise all network paths to a destination, notary data will let a client to detect that an attack is likely underway. *Temporal redundancy*, provided by the key history data returned by each notary, can offer additional protection because even if an attacker compromises all paths to the server, clients can still detect that a recent key change occurred and regard the new key with suspicion. Finally, *data redundancy* (described in Section 5) helps clients detect malicious notaries that supply inconsistent information, thereby limiting the effectiveness of attacks on the notary infrastructure itself.

The precise attack resistance provided PERSPECTIVES depends entirely on how a client’s key-trust policy sets the parameters defining spatial, temporal, and data redundancy in order to balance the risk of accepting an unauthenticated key with the possibility of incorrectly rejecting a valid key. Sections 4 and 6 explore this trade-off in detail.

2.2 PERSPECTIVES vs. a Standard PKI

At a high level, PERSPECTIVES might be described as a “lightweight PKI”. While both PERSPECTIVES and a standard PKI require that clients securely retrieve one or more public keys to bootstrap trust, there are two key differences between PERSPECTIVES and the traditional PKI currently used to grant SSL certificates:

1. **Mechanism for binding hostnames to public keys:** Traditional PKIs use an *offline* mechanism to determine that the real-world entity requesting a certificate in fact owns the associated hostname. PERSPECTIVES uses automated network probing to bind keys to a hostname.
2. **Degree of client control over trust decisions:** With a traditional PKI, the certificate authority makes a universal judgment regarding key validity. With PERSPECTIVES, each client independently interprets notary data and makes a decision based on its own security requirements.

Because probing by network notaries does not protect against all possible network attacks, we expect that highly sensitive services like bank or large e-commerce websites will continue to use heavyweight PKI mechanisms (possibly augmented with notary data, see Section 8). However, we believe that PERSPECTIVES provides a simple and cost-effective way to improve the attack resistance of services that currently either use Trust-on-first-use or are completely unauthenticated.

3 The Notary Architecture

We now explore how the notary infrastructure provides spatial and temporal redundancy to help clients evaluate an untrusted public key. We defer some implementation details until Section 7. *Notary servers* are a coordinated group of hosts distributed across the Internet. *Notary clients* are integrated into applications (e.g., an SSH client) and contact notary servers to download observed key data with which to make a key-trust decision.

3.1 Notary Administration

We envision a network notary group to be a fixed group of at least five (but possibly many more) servers located in diverse network locations. The group may be run by a single entity, but we design it in a decentralized fashion to also support a cooperative deployment in which universities, ISPs, hosting providers, or other well-known organizations each contribute one or two nodes.¹ Cooperative Internet testbeds like PlanetLab [18] and RON [2] are another attractive deployment option. For simplicity we describe a single notary group, but in practice multiple notary groups controlled by different entities could operate in parallel.

Each notary group is organized by a *notary authority* that determines which machines are legitimate notary servers. The notary authority has a public/private key pair and publishes its public key ($K_{Authority}$) using an out-of-band mechanism (e.g., as with Tor, the key could be distributed with any software that accesses the notary group). To add a notary server X to the system, X ’s owner generates a public/private key pair and furnishes the public key K_X and its IP address A_X to the notary authority. Each day the notary authority publishes a list of notary server IP address (A_i) and public key (K_i) pairs to each of the notary servers in the systems, along with a signature $S = \{Date, (A_1, K_{A_1}), (A_2, K_{A_2}), \dots, (A_n, K_{A_n})\}_{K_{Authority}^{-1}}$. A client can contact any notary server, download the list, and

¹Several existing projects have successfully used a similar decentralized deployment model, including the large collection of publicly available traceroute and looking-glass servers [25], the Tor anonymizing network [8], and the NTP Pool’s large set of publicly available network time sources [16].

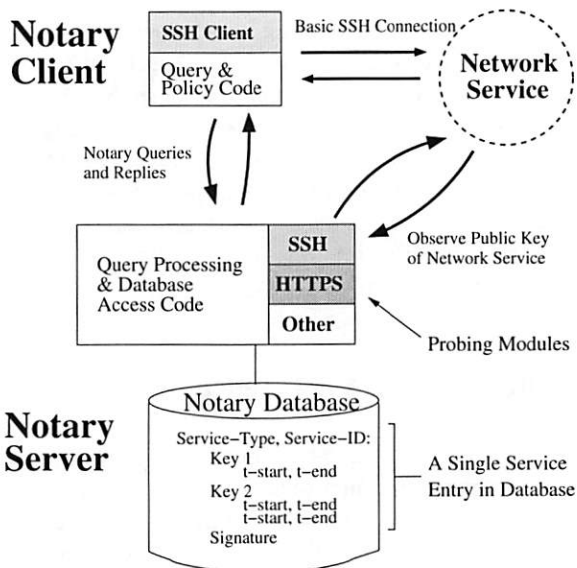


Figure 2: Schematic of a notary server and SSH client.

validate this signature using $K_{Authority}$ to receive a fresh list of notary IP addresses and public keys.

3.2 Notary Server Key Monitoring

Notary servers monitor the public key(s) used by a network service over time. A notary server provides clients with an application-independent query interface, but uses application-aware *probing modules* to monitor different types of services (e.g., SSH or HTTPS). A probing module observes keys by connecting to the service and mimicking an ordinary client until it receives the service's public key, at which point it disconnects.

Each notary server uses a local database to store a *service entry* for each monitored service. A service entry contains all observed key data the notary has recorded while monitoring that service over time (see Figure 2). An entry is uniquely identified by the combination of a *service-type*, which identifies the protocol used to retrieve the key (e.g., HTTPS)² and a *service-id*, which provides the information necessary to contact the service (e.g., host-name and port). The observation history of each key is stored as one or more *key timespans*. A key timespan is a start and end timestamp pair (t_{start} , t_{end}) that indicates a period of time during which the notary observed only that key for the service. When the notary makes a new observation it updates the corresponding service entry in the following manner: if the observed key is the same key observed during the previous observation, the notary simply updates the t_{end} value of the most recent key timespan to the current time. Otherwise, the notary creates a new

² A single logical protocol may have multiple service-types (e.g., an SSH2 server can have both RSA and DSA keys).

Notary Protocol:

$C \rightarrow N$: $S = (\text{service-id}, \text{service-type})$
 N : $(O, \{O\}_{K_N^{-1}}) = \text{find_service_entry}(S)$
 $N \rightarrow C$: $O, \{O\}_{K_N^{-1}}$

Figure 3: The basic notary protocol between a single client (C) and the notary server (N).

key timespan with both t_{start} and t_{end} set to the current time and adds this timespan and the new key (if necessary) to the service entry. If a probe fails to receive a key from the service, the notary creates or updates a timespan with a “null key” containing no key data.

The notary also stores a cryptographic signature for each service entry in the database. Using its private key, the notary calculates a signature over all data in the service entry: the (service-id, service-type), as well as each key and its associated key timespans. This signature is updated following each modification of the service entry. Section 7 measures the overhead of this simple signature scheme and mentions potential optimizations.

3.3 Querying Notary Servers

The client application contacts the notary whenever it receives a key from a service that does not match an existing entry in its cache. This may occur because the client has never contacted the service or because the offered key does not match the already cached key for the service.

When contacting an individual notary (Figure 3), the client specifies a (service-type, service-id) pair. The notary finds the corresponding entry in its database and replies with observed key data consisting of keys and their associated timespan(s), along with a signature over that data using the notary's private key.

The process by which a client queries for and receives notary data is described in Figure 4. Recall that a notary client learns about all notary servers and their public keys using a list distributed by the notary authority. The client's key-trust policy (Section 4) determines n , the number of notaries that the client should contact. The client then randomly chooses n entries from the list of notaries and queries these servers in parallel using UDP.³ The querying process is complete once enough notaries have replied for the client policy to make a trust decision, or when the client determines that all remaining notaries are unreachable (clients implement a simple retransmission strategy). The client validates the signature for each response using that notary's public key, discarding any invalid responses.

³ Using the implementation parameters described in Section 7, a 1460 byte MTU-sized UDP datagram can hold a notary reply with 44 key/timespan pairs. If necessary, multiple UDP packets are used for large replies to avoid IP fragmentation.

```

Check-Unauthenticated-Key( $s, K_{offered}$ )
 $O = \{\}$  // all observations for services
All-Notaries = load_notary_addresses()
Chosen-Notaries = choose_random( $n$ , All-Notaries)
foreach  $x$  in Chosen-Notaries in parallel
    ( $O_x, \{O_x\}_{K_x^{-1}}$ ) = contact_notary( $x, s$ )
     $K_x = \text{load\_notary\_key}(x)$ 
    if (verify_signature( $O_x, \{O_x\}_{K_x^{-1}}, K_x$ ))
         $O = O \cup O_x$ 
if ( ! check_trust_policy( $K_{offered}, O$ ) )
    abort_connection()

```

Figure 4: How a client checks an unauthenticated key $K_{offered}$ for service s . Details of *check_trust_policy*() are discussed in Section 4.

4 Notary Client Key-Trust Policies

Once a client has received observed key data from notary servers, it uses a *key-trust policy* to accept or reject an offered key based on this data. Code implementing the client policy decision examines the offered key and all validated observed key data, and may also consider inputs such as previously cached keys, user security preferences, or even active user input. Upon completion, the client application either accepts the key and continues running the protocol or rejects the key and disconnects.

4.1 The Security vs. Availability Trade-off

The SSH model (which includes both Trust-on-first-use and PERSPECTIVES) presents clients making a key-trust decision with a basic *security vs. availability trade-off*: faced with an untrusted key, the client can take a security risk and accept the key, or be safe and reject the key, at the cost of making the service (at least temporarily) unavailable. For example, when faced with an unauthenticated key, standard Trust-on-first-use makes two different security vs. availability trade-offs: First, if no key is cached for the service, Tofu chooses availability at the cost of security by always accepting the offered key. Second, in the case of a key conflict, a Tofu application cannot automatically accept the key, so it must favor security over availability or prompt the user for help.

Lacking useful information about the key's validity, Tofu applications are stuck making this inflexible trade-off. In contrast, because PERSPECTIVES provides additional data indicating whether a key is likely to be the result of an attack, it allows application key-trust policies to make significantly more intelligent security vs. availability trade-offs.

In this section we explore several variations on client key-trust policies. We do not claim that these policies are optimal (in fact, significantly more nuanced and complex policies exist), but rather offer them as evidence that even

simple policies can support a wide range of security vs. availability trade-offs.

4.2 Quorum: A Key-Trust Primitive

Recall that PERSPECTIVES provides security by allowing clients to leverage spatial redundancy (key observations from multiple vantage points) and temporal redundancy (key observations over time). Therefore, the role of a client policy is to test the spatial and temporal consistency of the offered key with respect to notary data. To provide a framework for reasoning about spatial and temporal consistency, we introduce threshold parameters that quantitatively represent these properties.

Definition: For a set of n notary servers, a service S , and a threshold q ($0 \leq q \leq n$) we say that a key K has *quorum* at time t iff at least q of the n notaries report that K is the key for S at time t .

Intuitively, for values of q that are large relative to n , a key that has quorum indicates consensus among the observations made by the all notaries at a single point in time. We use another threshold parameter to extend the concept of quorum into the temporal realm.

Definition: For a set of n notary servers, a service S , and a quorum threshold of q , a key K has a *quorum duration* of d at time t iff for all t' such that $(t - d) \leq t' \leq t$ the key K had quorum with threshold q at time t' .

Quorum duration indicates how long, without interruption, a set of notaries has consistently seen a particular key.⁴ Applications can make security vs. availability trade-offs by choosing to accept an untrusted key only if it exceeds a particular quorum duration threshold. Higher q and d thresholds provide more security, but risk reducing availability by incorrectly rejecting valid keys. For example, setting q equal to the total number of notaries n provides the strongest protection against accepting a false key, but also means that a single unavailable or compromised notary could cause the client to reject a valid key. Similarly, a higher quorum duration threshold d protects against a strong attacker that compromises many paths for a significant amount of time, but would also require clients to reject connections to services that are new or have recently changed keys.

Next, we consider three examples of how the concepts of quorum and quorum duration might be used by a PERSPECTIVES client policy.

⁴ We intentionally do not require that the quorum be comprised of a stable set of notaries over the entire duration.

Example Policy: Expert User

Same number of warnings, more useful data.

We first consider a client application policy for a “expert user” who understands the risk of a MitM attack and is familiar with the SSH authentication model. In this case, the client always warns the user but also includes concise summaries of the notary data that help the user make a better security vs. availability decision. Similar to existing Tofu client, policy behavior depends on whether the offered key conflicts with an existing cache entry.

Case 1: No Server Key Cached

In this case, the user is not necessarily suspicious of the new key, but she will use observed key data to confirm that the key is consistent across many notaries and that the duration of the key history is commensurate with her expectation of key age. If quorum duration is satisfied, the policy module may supply a message like:

Key seen consistently for the past Z days.

If the key fails to achieve quorum or does not have sufficient quorum duration based on policy parameters, the user might see one of the following warnings:

SUSPECTED ATTACK: Offered key is NOT consistent.
Only X of Y notaries currently see it.

WARNING: Server key has only been seen
consistently for the past Z days.

Case 2: Offered Key Differs from Cached Key

In this case, the user must distinguish between a legitimate server key change and a falsely injected key. As in Case 1, the user will be interested in the prevalence and duration of the offered key, since a substantial quorum duration indicates a higher likelihood that the conflicting key is the result of a legitimate server key change:

Offered key conflicts with cached key, but has
been consistently seen for Z days.

Additionally, because the policy has access to the cached key that is currently trusted for the server, the policy might also highlight portions of the key history that cast suspicion on the new key. For example, a warning might indicate that some notaries are still currently observing the cached key, suggesting that the server has not actually changed its key:

LIKELY ATTACK: Offered key conflicts with
cached key and cached key is still observed
by X of Y notaries!

Interpreting such statements requires little additional work

on the part of the user but provides her with vastly more information than warnings in current Tofu applications like SSH. In the rare case that such a summary is insufficient, expert users may view all observed key data, similar to how web browsers optionally display SSL certificates.

Example Policy: Non-Expert User 1

Same number of warnings, varied severity.

Because non-expert users are unlikely to want or be able to make good security decisions based on the notary data itself, another policy approach is use a quorum duration test to determine how severe of a warning to give the user. For example, a simple approach would be to give users a standard warning as long as three-quarters of the notary servers have seen the key for at least a day (i.e., $q = 0.75 \cdot n$ and $d > 1$ day), but give them a more severe and intrusive “security failure” warning if notary data detects a key inconsistency indicating a probable attack. .

For HTTPS the need for such a policy to distinguish likely attacks from valid self-signed replies is increasingly evident, as new versions of two major browsers (Internet Explorer 7 and Firefox 3) have introduced new user interfaces that treating self-signed certificates as failures by default. Rather than simply displaying a warning dialog, the new interfaces do not render the page at all and instead display an error page similar to a failed connection.

Example Policy: Non-Power User 2

Fewer warnings, based on high-level preferences.

Much usability research suggests that web users often make bad security decisions by ignoring warnings [21, 10]. Instead of using quorum duration to determine the severity of a warning, a client could choose to issue no warning at all if an offered key has sufficient quorum duration. The precise values for q and d may be determined by high-level user preferences for “high security” or “medium security” already common in browsers today.

While the merits of this approach is ultimately a usability question, this tactic may help reduce “banner blindness” associated with browser security warnings. As an additional benefit, this approach could increase overall adoption of HTTPS by eliminating the frequent security dialogs that likely make some website owners hesitant to use self-signed certificates at all.

5 Detecting Malicious Notaries

The final aspect of the notary design is *data redundancy*, a cross-validation mechanism that limits the power of a compromised or otherwise malicious notary server.

To implement data redundancy each notary acts as a *shadow server* for several other notaries. As described below, a shadow server stores an immutable record of each observation made by another notary. Whenever a client

receives a query reply from a notary, the client also checks with one or more of that notary's shadow servers to make sure that the notary reply is consistent with the history stored by the shadow server.

5.1 Benefits of Data Redundancy

An adversary in control of a notary and its corresponding private key can provide clients with false observed key data. Data redundancy prevents a notary from changing data already recorded in its observation history, much as schemes such as forward-secure signatures [3] do. As a result, an attacker that compromises a notary cannot, for example, create a new malicious key and falsely claim that this key has been stably seen over a long period of time. This cross-validation ensures that the only way an adversary can make a malicious key appear "stable" is by sustaining a network attack for a commensurate amount of time, even with notary compromises. Additionally, data redundancy guarantees that a notary, even after compromise, cannot give conflicting answers to two clients querying about the same service. This property (which cannot be achieved using forward-secure signatures) could help hosts scan for and detect notaries that act maliciously.

5.2 Cross-Validation Protocol

Each entry in the list published by the notary authority also lists MAX-REDUNDANCY other notaries that each act as a shadow server for the notary specified in the entry. A notary server is responsible for keeping all of its shadow servers up-to-date. When a notary server contacts a service S , it updates its local database and then sends the new service entry (including signature) to each of its shadows.⁵

Shadow servers update their shadow copies in a way that prevents malicious notaries from eliminating previously shadowed data. To do so, the shadow server requires that each key timespan in the old shadow copy either also exists in the new shadow copy or is "contained" within a larger timespan in the new copy (i.e., both timespans have identical t_{start} values, but the new copy's t_{end} value is greater than that of the old copy). Additionally, no timespans in the new copy can overlap. If the old and new data are consistent, the old data is discarded. If an inconsistency exists, the shadow server stores both sets of observed key data and signatures. After updating, the shadow server uses its own private key to generate a signature over both the shadowed data and the signature of the other notary. This signature is stored with the shadowed data and allows clients to authenticate that a reply to a shadow request came from the correct shadow server.

Client policy specifies r ($r \leq \text{MAX-REDUNDANCY}$), the number of shadow servers that must corroborate a

⁵Client consistency checks are specifically designed so that a malicious notary does not benefit by failing to keep its shadows up to date.

Cross-Validation Protocol:

$C \rightarrow SH :$	N, s
$SH :$	$DB_n = \text{get_replica_database}(N)$
$SH :$	$O_N, \{O_N\}_{K_N^{-1}}, \{O_N, \{O_N\}_{K_N^{-1}}\}_{K_{SH}^{-1}}$ $= \text{find_service_entry}(DB_N, s)$
$SH \rightarrow C :$	$O_N, \{O_N\}_{K_N^{-1}}, \{O_N, \{O_N\}_{K_N^{-1}}\}_{K_{SH}^{-1}}$

Figure 5: A client (C) contacting shadow server (SH) for a shadow copy of notary N 's observed key data for service s . Note that the integrity of N 's service entry is protected in transit by a signature using the shadow server's key.

notary's observation for it to be deemed valid. For each notary N the client contacts, it randomly selects r_q ($r_q \geq r$) of N 's shadow servers to query. When contacting a shadow server (Figure 5), the client specifies the IP address of N as well as the service-id from the original query to N . The shadow server replies with a service entry (observed key data and signature) created by N , along with the shadow server's signature over that data. If fewer than r of N 's shadows provide valid responses signed by K_n , the client disregards all data from N . After verifying the signatures, any inconsistencies among the original and shadowed data will cause the client to reject data from N . Additionally, because of non-repudiation, clients can provide any inconsistent data and signatures to the notary authority as evidence of a notary's malicious behavior.

6 PERSPECTIVES Security Analysis

To demonstrate the benefits of network notaries, in this section we enumerate realistic attack scenarios and compare the security provided by PERSPECTIVES to that offered by basic Trust-on-first-use. We analyze both an adversary's ability to launch a MitM attack, as well as its ability to deny availability (DoS) by causing a client to reject a valid key.

MitM Attacker Resources:

Our analysis considers three possible system components which an adversary may control. Enumerating each combination of these possible compromises lets us analyze all scenarios relevant to the attack resistance of PERSPECTIVES:

- L_{client} : An adversary controlling the client's local link can modify or drop all client-to-service and client-to-notary communication.
- L_{server} : A compromise of the server's local link lets an attacker inject arbitrary keys when either clients or notaries contact the server.
- $k \cdot n_m$: A compromise of k distinct notary servers.

Compromise	Tofu		PERSPECTIVES	
	DoS	MitM	DoS	MitM
L_{client}	✗	✗	✗	safe
L_{server}	✗	✗	✗	temporal safe
$k \cdot n_m$	safe	safe	$k > (n - q) : \text{✗}$ $k \leq (n - q) : \text{safe}$	safe
$L_{server} + L_{client}$	✗	✗	✗	temporal safe
$L_{client} + k \cdot n_m$	✗	✗	✗	$k \geq (q + q \cdot r) : \text{✗}$ $k \geq q : \text{temporal safe}$ $k < q : \text{safe}$
$L_{server} + k \cdot n_m$	✗	✗	✗	$k \geq (q + q \cdot r) : \text{✗}$ $k < (q + q \cdot r) : \text{temporal safe}$
$L_{server} + L_{client} + k \cdot n_m$	✗	✗	✗	$k \geq (q + q \cdot r) : \text{✗}$ $k < (q + q \cdot r) : \text{temporal safe}$

Table 1: Summary of attack resistance provided by PERSPECTIVES in comparison to the standard Tofu approach. The left column contains abbreviated attack descriptions as defined in the text. Columns show resistance to availability (DoS) and MitM attacks. ✗ indicates no resistance, “safe” indicates that attacks are detected, and “temporal safe” indicates temporal safety, as defined below.

As discussed earlier, L_{client} is a likely attack, due to the prevalence of open wifi hotspots and insecure home networks that allow compromised hosts or home routers to easily inject or modify traffic. In contrast, servers often reside in more controlled network environments, making L_{server} significantly harder. This is particularly true in the case of HTTPS. Thus, to achieve L_{server} , an adversary might compromise the gateway router for the destination, or use BGP to falsely announce the destination’s prefix, misdirecting some or all traffic destined for the server. An attacker may also control arbitrary Internet routers that place it on k of the n notary-to-service paths. However, because this attacker is a strictly weaker version of the $k \cdot n_m$ attacker, we do not examine it separately. Additionally, since L_{client} or L_{server} allows the attacker to mount a trivial DoS attack in either scenario, we do not explicitly mention this attack below.

Analysis Parameters:

As described in Section 4, the actual security and availability that a client will receive depends on its choice for the following policy parameters:

- n:** Number of notary servers contacted by the client.
- q:** Quorum threshold of client.
- r:** Number of shadow servers (per notary) the client requires for data redundancy.

We do not directly model the length of an attack or a client’s quorum duration. Instead, we use the concept of “temporal safety”, which means that a client will be safe as long as its quorum duration threshold is larger than the ac-

tual duration of the attack.⁶ Table 1 summarizes the results.

Analysis Results:

L_{client} Compromise: When only the client’s access link is compromised, Tofu provides no defense against a MitM attack, while data from network notaries allows a client to easily detect and avoid the same attack.

It is important to recognize that the attacker gains no MitM advantage by using L_{client} to disrupt client-to-notary communication. All data returned by a notary is protected by a signature from that same notary, meaning that notary responses cannot be spoofed, even if the attacker has compromised other notaries. Furthermore, adversaries cannot encourage acceptance of a false key by making notaries or shadow servers unreachable, since blocking such communication will prevent any key (including a malicious one) from achieving quorum. Maliciously dropping client-to-notary communication to prevent a legitimate key from achieving quorum does not increase attacker power, since control over L_{client} already allows for a trivial DoS attack by simply dropping all packets.

L_{server} Compromise: A compromise of only the server link also renders a basic Tofu client vulnerable to MitM attacks. For PERSPECTIVES, the compromise of all paths to the server will prevent spatial diversity alone from detecting an attack. However, historical key data provides a client with temporal safety against network attacks.

⁶Additionally, to simplify our analysis, we assume a sufficiently large number of notary servers such that no overlap exists among the q notary servers and the $q \cdot r$ shadow servers used by a client. A set of notaries that is too small to satisfy this assumption would reduce the number of compromised notaries needed to undermine data redundancy.

$k \cdot n_m$ Compromise: The compromise of notaries alone will not enable an adversary to inject a false key to the client and launch a MitM attack. However, this adversary can attack availability by trying to cause a client to incorrectly reject a valid key. Disabling $k > n - q$ notary servers, either by compromising them or by making them unreachable, prevents the client from establishing a quorum even if the remaining servers all agree on the valid public key. Because this attack is possible even if the adversary is not on the client-to-service path, it represents an availability vulnerability that does not exist with the Tofu approach. However, this attack is limited to scenarios when clients receive a new key for a service; it does not apply to repeated connections between a client and a server using a cached key.

$L_{client} + L_{server}$ Compromise: The analysis of this scenario is identical to L_{server} , since, as discussed above, using L_{client} to restrict client access to notary servers provides no attack benefits.

$L_{client} + k \cdot n_m$ Compromise: Control over the client link and some notary nodes lets the attacker use notaries to “promote” an invalid key using false observations. An attacker cannot perform a MitM attack unless it compromises a full quorum q of notaries, since the client rejects keys that do not achieve quorum. If an attacker compromises q notary servers, the situation is identical to the L_{server} scenario described above: the client is still protected for a time period determined by its quorum-duration. However, if the adversary compromises an additional $q \cdot r$ notaries beyond the basic quorum, it can overcome the data redundancy of the system. Without data redundancy the attacker can forge the observation history, eliminating the protections of temporal safety.⁷

$L_{server} + k \cdot n_m$ Compromise: This attack is stronger than the previous $L_{client} + k \cdot n_m$ scenario, since control over the destination service’s link means that even legitimate notaries will observe the attacker’s key. As a result, even if fewer than q notaries are compromised, the client relies entirely on temporal safety.

$L_{server} + L_{client} + k \cdot n_m$ Compromise: This scenario is identical to the previous attack. As described in the $L_{server} + L_{client}$ case, client link access grants no additional power if an adversary already has server link access. This attack analysis demonstrates that PERSPECTIVES significantly improves resistance to MitM attacks compared to Tofu. Additionally, PERSPECTIVES is robust to limited compromises of the notary infrastructure itself.

⁷We note that this is a worst-case analysis. If the client selects notaries randomly and the total number of available notaries is larger than $(n + n \cdot r)$, the attacker cannot easily predict which notaries or shadows it must compromise in order to mislead the client.

7 Experience with Notary Server and Client Implementations

To demonstrate the viability of a network notaries and to gain experience with its deployment, we have implemented and are running a publicly available network notary on the RON testbed [23]. Additionally, we have created two different PERSPECTIVES clients: a modified version of the OpenSSH client for SSH and an extension to the Mozilla Firefox browser for use with HTTPS certificates. We have made both server and client source code publicly available at: <http://www.cs.cmu.edu/~perspectives/>. Our performance measurements indicate that a single notary server can monitor several million hosts a day while simultaneously handling a large number of client queries.

7.1 Notary Server Implementation

Our notary server code is written in C and uses the Berkeley DB library for storing observed key data and signatures. The notary probes services running either SSL or SSH using probing modules based on code from OpenSSL and OpenSSH. The only substantial difference between our implementation and the design described in Section 3 is that we do not implement data redundancy, in part because our notary deployment is run by a single trusted entity.

We benchmarked our notary server with respect to service monitoring and query response operations to demonstrate that network notaries are practical on commonplace hardware. In our implementation, notaries use 1369-bit RSA keys for service entry signatures⁸ and store public keys as 128-bit MD5 fingerprints.⁹

We run our each benchmark on two different machines. One is a modern server (*ServerFast*) with two dual-core 2 GHz AMD Opteron CPUs and 8 GB of RAM. The other server (*ServersLow*) is a three year old machine with a single-core P4 2.4GHz CPU and 512 MB of RAM, intended to demonstrate that a modest notary infrastructure could still be run on older (perhaps donated) hardware. A summary of the results is provided in Table 2.

Monitoring Load: To benchmark notary monitoring, we identified a set of .com domains running HTTPS using a web-crawl and had our notary server monitor these sites (Our SSH scans exhibited nearly identical rates and are therefore omitted). When monitoring a service, the notary must perform the protocol negotiation necessary to retrieve the service’s key, load the existing service

⁸ Notary signatures covering observed key data do not need long-term security, as signatures are recomputed frequently and notary keys are easily updated. 1369-bit RSA is deemed secure through 2010[14]. Only $K_{Authority}$ needs long-term security (e.g., 2048 bits) to enable key rollover.

⁹While MD5 collision resistance has been compromised, the security of public key fingerprints depends instead on second pre-image collision resistance, which is still considered secure for MD5.

Operation	Operations / Sec	
	<i>Server_{Slow}</i>	<i>Server_{Fast}</i>
Monitor service key	26	195
Monitor service key (no sig.)	112	270
Handle query (in memory)	21,700	25,600
Handle query (disk-bound)	114	-

Table 2: Summary of notary server benchmarks for machines *Server_{Slow}* and *Server_{Fast}*.

entry from the database, update the entry, recompute the entry's signature, and store the entry again. Our code uses RSA signatures implemented in OpenSSL and takes the simple (albeit heavyweight) approach of forking a process running a slightly modified OpenSSH or OpenSSL client for each probe.¹⁰ *Server_{Fast}* and *Server_{Slow}* performs 195 and 26 such operations per second, respectively. While these rates may seem small, even at only 50% utilization they correspond to 8.4 million and 1.1 million probes per day. Table 2 also shows monitoring rates when signatures are not computed, indicating that optimizing the cryptographic processing by batching service records or creating a hash-tree over multiple service entries would significantly improve performance.

Query Handling Load: The primary consideration for query processing performance is whether the notary's databases fit in memory (recall that a notary may have multiple databases due to shadowing). With the cryptographic parameters described above, a single service entry with several keys and timespans will consume approximately 250 bytes. Thus, a database with one million entries (250 MB) can easily fit even in the small memory of *Server_{Slow}*. Table 2 shows benchmarks for both servers responding to randomized client queries (measured over the loop-back interface). If the database is in memory, the response rate is above 20,000 requests/sec for both servers (because our server code is single-threaded, there is little difference between the two machines). To identify likely response rates when the database must be accessed from disk, we tested *Server_{Slow}* with a database that was four times the size of its main memory. In this case, the speed of a disk-seek limits the response rate to 114 queries/second. However, even this query-processing rate (which translates to just under 10 million per day) may be viable when clients contact notaries only on rare cache misses.¹¹

Bandwidth Requirements: Bandwidth usage depends on

¹⁰OpenSSH's *key-scan* utility served as our initial SSH scanner, but it exhibited bugs that caused us to discontinue its use.

¹¹This disk-bound query-handling does not significantly contend with the CPU-bound service monitoring, meaning that their respective rates are unlikely to diminish significantly when both are run simultaneously.

both the monitoring rate and query rate. Including all network headers and TCP acknowledgments, monitoring a single SSH service requires about 1.5 KB of upstream bandwidth and 2.3 KB of downstream bandwidth. For SSL the same values are 0.5 KB and 2.0 KB. Such data sizes are comparable to a single small image embedded in a webpage. Monitoring a million hosts a day would correspond to an average rate of less than 213 Kb/s in each direction. With network headers, notary requests are approximately 60 bytes and replies are 315 bytes (for standard key sizes). While handling a flood of 20,000 requests per second would require 50 Mb/s of upstream bandwidth, a server could handle 10 million requests a day (116 requests per second) using only 420 Kb/s.

Because we expect notary nodes to be deployed primarily by universities and large companies like ISPs or webhosting providers, we do not expect bandwidth to be a limiting factor unless a server is under DoS attack.

7.2 Notary Client Implementations

To demonstrate the general nature of the PERSPECTIVES approach, we created two client implementations that access our notary group. Both clients share common library code that reads notary configurations, creates and parses protocol messages and implements basic quorum duration policies. As a result, we expect that porting other clients to work with PERSPECTIVES will not be difficult.

OpenSSH Client: We modified the popular OpenSSH [17] client software to contact notary servers when no cached key exists or when the offered key differs from the cached key. The modifications consisted of a few library calls within the SSH code that checks the key cache, as well as functionality to interact with the user when making a key-trust decision (our client roughly follows the expert user example described in Section 4). We have used the modified client within our university for nearly a year and have found the added latency of contacting notary servers to be negligible.

Mozilla Firefox HTTPS Client: We implemented our notary client for HTTPS within the Mozilla Firefox extension framework, meaning that users can easily install it without having to download a new browser. The HTTPS notary extension (Figure 6) is written in a combination of C++ and javascript. Whenever a certificate validation fails, the notary client extracts the public key from the certificate and contacts the notaries.

Our implementation allows users to set a preference for "High Security" or "Medium Security", which correspond to different quorum duration thresholds. Keys that achieve quorum duration are accepted automatically, while other keys result in disconnection with an error message. The "High Availability" security setting gives expert users the

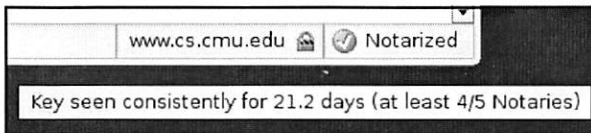


Figure 6: This screen capture shows the bottom right-hand corner of a Firefox browser window, which contains the HTTPS notary extension’s visual indicator. In this case, the website satisfied quorum duration, so the browser automatically accepted the certificate and suppressed the certificate warnings usually displayed to the user. The image also shows the tool-tip that summarizes the notary data for the user.

ability to connect even when a key does not achieve quorum duration. These users are provided with the certificate and a summary of notary data in order to make their own key-trust decision.

Because use of our OpenSSH and Firefox clients is currently limited to the Linux and FreeBSD/Mac OS X platforms, we also provide a web page that allows users to query notary servers even without a modified client.

7.3 Generating a List of Services to Monitor

Section 3 assumes a priori knowledge of what services the notary server should periodically monitor. We consider two possibilities for how notaries might build this list.

The first approach, which we use in our implementation, is for a notary to add a service to its database the first time a client queries for it. While this approach does not initially provide clients with temporal redundancy, when bootstrapping the system, notaries might probe “on-demand” to provide even the first client with spatial redundancy to help detect L_{client} attacks. A client could seed the notary with services (e.g., from the `known_hosts` file in SSH) and a server could register for monitoring by querying a notary with its own address as the service-id.

The second approach is to proactively discover services. For example, HTTPS websites can be found using web crawling or search engines. For less public services, such as SSH, TCP-layer scanning on standard protocol ports could discover a large number of services. We built one such scanning engine to evaluate this approach on SSH servers within our campus network and a few large public IP blocks. We found that while scanning can quickly identify tens of thousands of SSH servers, it has three key limitations. First, if reverse DNS is unavailable for an IP address, we cannot reference a service entry in terms of the DNS name. Second, such scanning misses services running on non-standard ports. Finally, scanning may be misinterpreted as an attack. Because active discovery cannot identify all services, one promising approach is to seed notaries with data from active discovery, and then identify additional services as clients query for them.

7.4 Notary Parameters

We now outline the primary considerations for choosing a notary group’s global parameters.

Number of Notaries: Because MitM attacks are most likely at the network edge (i.e., L_{client} or L_{server}), increasing spatial redundancy is likely to have diminishing returns: in the case of L_{client} , even a few valid notaries will detect an attack, while for L_{server} , all notaries will see the same false key. We therefore suggest that clients query from 4 to 10 notaries, depending on their desired robustness to notary compromises or failures.

Notary Monitoring Frequency: A notary would like to minimize the time between when a server first comes online or changes its public key and when the notary server observes this change. Our measurements indicate that a single notary could monitor several million different servers a few times each day.

Degree of Data Redundancy: The degree of data redundancy required for a set of notaries depends greatly on who administers individual notary nodes. If nodes are run by one or a few trusted entities that take great care to secure the machines, little or no data redundancy may be needed. However, even when data redundancy is needed, MAX-REDUNDANCY can likely be small (2 to 4), because clients can detect inconsistencies if any one of the contacted shadow servers are not compromised.

8 Discussion

Notaries and DNS Attacks: So far, we have focused on adversaries that compromise IP-level paths or notary servers. Adversaries could also manipulate DNS to falsely map a service’s hostname to an IP address that places the adversary “on path”. When notaries and clients use DNS names to identify services, a compromised local or remote DNS server present the same threat as the corresponding L_{client} or L_{server} attack. Thus, the analysis in Section 6 applies to DNS attacks as well.

Additionally, if an attacker controls only DNS, notaries can help clients detect and even circumvent such attacks. To do so, notaries would also record the IP address used when monitoring a service. If a client sees that both the service key and IP have changed from a previously trusted or stable key, it can connect directly to the IP addresses associated with the old key to test if that key is still visible on the network. If the client receives the prior key from any of the past addresses, it can (depending on local policy) disregard the new key as a DNS attack and instead

connect directly to the address using the prior key.¹²

Client Privacy Considerations: While benign notary servers would not record a client's IP address, a malicious notary could link client addresses and destination services, impinging on client privacy. Because clients access the notary group only rarely (when the offered key is not cached) and do so only when a legitimate security threat exists, we believe that most clients will consider this potential privacy risk acceptable.

However, clients that desire additional privacy could contact the notary group through a proxy. One promising design for such a proxy is to have the notary authority run a DNS nameserver for a special notary domain (e.g., *notary.com*). A notary client looking for observed key data for *server.domain.com* from *num-notary* different servers would perform a DNS look-up for *num-notary.server.domain.com.notary.com*.¹³ The nameserver for *notary.com* would then randomly query *num-notary* servers and return the base64-encoded results as a DNS TXT record. Because of the recursive nature of DNS look-ups, the notary nameserver would learn only general information about the client (e.g., that they use CMU's DNS servers), and the rest of the notaries would learn no client-specific information. Clients are unlikely to have privacy concerns about notary queries via local DNS resolvers, since they already expose basic connection information to the resolver with standard DNS look-ups.

Detecting Authentication Downgrade Attacks: Our primary motivation for designing PERSPECTIVES was to help authenticate services that do not have certificates signed by a global PKI. However, users can also benefit from using PERSPECTIVES even when accessing websites with PKI-signed certificate by gaining protection against *authentication downgrade attacks*. In an authentication downgrade attack, a MitM adversary injects a self-signed certificate in place of the PKI-signed certificate sent by the legitimate server. Because the attacker can spoof legitimate names in the domain name and issuer fields, the significant number of users who routinely ignore many browser security warnings [21, 10] would fall victim to such an attack. In fact, a malicious exit-node in the Tor anonymizing network was recently observed running such an attack [24]. However, if notaries also monitor services with PKI-signed certificates, clients could detect this attack by comparing the received certificate against the notary replies. This same approach would also help prevent the even more damaging attack in which an

attacker tricks one of the many root CAs trusted by a browser into signing an invalid certificate.¹⁴

On-demand Service Monitoring: A slightly modified notary design could allow clients to request that notaries probe a service "on-demand", for example, when the client's offered key does not match the most recent entry in the notary history. Depending on client policy, this approach could reduce the likelihood that a client incorrectly rejects a key following a legitimate server key change. However, because the notary must cryptographically sign the each new probe result, it would be more vulnerable to DoS attacks. In light of this and other potential abuses, on-demand probing is best suited for either a private notary group with limited access or a public notary group augmented with a strong rate-limiting mechanism like client puzzles [13].

Scaling the Notary Infrastructure: Our design and implementation focuses on a notary infrastructure that is easily deployed and capable of regularly monitoring several million unique services. If PERSPECTIVES is widely adopted as a standard host authentication mechanism, it can easily scale. First, note that notary replies are simply static data, which could be made available via a content delivery network (e.g., Akamai) or a network storage service (e.g., Amazon's S3). The physical location or ownership of these machines would be unrelated to the hosts generating observed key data. Because monitoring and updating observed key data is trivially parallelizable, this work can be distributed to a cluster of machines in each notary location. Trends toward many-core machines should further improve the efficiency of service monitoring.

9 Related Work

Significant work exists on the problem of authenticating remote Internet hosts. Standard solutions include X.509 certificates within a global PKI [5], or Kerberos [15], which assumes that each participant has a shared secret with a trusted third party. Such solutions are extremely useful, but the popularity of the SSH model demonstrates the need for lightweight alternatives.

Ali and Smith [1] propose improving SSH key authentication using a "portable key cache", which the user stores along with an authenticating MAC on a personal webserver. With a modified SSH client, the user can access this cache from any machine, use a passphrase to verify the integrity of its contents, and then compare the offered key to entries in the cache. This design helps users who would otherwise see the same new or changed key warning several times

¹² Adding IP addresses to the observed key data returned by a notary also helps clients handle cases when DNS-based load-balancing maps a single hostname to different machines that each have their own key.

¹³ A similar DNS trick is used by the popular Coral Cache content distribution network [11]. Using a TTL of 0 and appending random data to the beginning of the DNS name can prevent DNS caching from providing stale data.

¹⁴ To accommodate legitimate key turnover, the site owner can sign the new public key using the older key that is already recognized as valid by the notaries.

when connecting to the same server(s) from multiple client machines. However, unlike PERSPECTIVES, it provides no help in determining a key's validity when a user either accesses a service for the first time or when an offered key does not match the key in the user's portable cache.

Advocating a more significant departure from the standard SSH authentication model, RFC 4255 [22] proposes storing SSH host key fingerprints in DNS. Unfortunately, this proposal relies on the deployment of the secure DNS PKI to authenticate the fingerprint data itself, and secure DNS has shown little traction to date. Additionally, the proposal requires that a domain's DNS administrator fulfill the responsibilities of a certificate authority: verifying that a real-world entity who contacts him with a certificate request legitimately owns a particular host. In contrast, PERSPECTIVES requires no heavyweight verification and instead automatically creates authentication data using probes.

ConfIDNS [19] suggests performing DNS look-ups from diverse network vantage points. However, the primary focus of the ConfIDNS work was dealing with the fact that DNS replies (unlike public keys used in PERSPECTIVES) frequently have legitimate inconsistencies due to factors like DNS load-balancing. Additionally, because the system was designed to avoid pollution in cooperative DNS systems, the design only protects against a malicious or failed *local* DNS server, not an "on path" adversary (e.g., *L_{client}*) launching a MitM attack.

Web tripwires [20] are javascript verification functions embedded by a webserver in HTTP responses to perform client-side detection of in-flight data modification. While lightweight, tripwires are not robust to adversaries, which can remove the tripwire code from the HTTP response to thwart detection.

Notary client policies bear some similarity to client behavior in the "web-of-trust" model used by Pretty-Good-Privacy (PGP) [4], a decentralized PKI for email. However, because PGP uses human contact to bind entities to keys, its primary challenge is estimating the strength of key trust chains that include multiple links, each representing a pair of real-world acquaintances. PERSPECTIVES policies do not have trust chains (each notary probes a service directly), but do have other complexities not seen in PGP, including the temporal nature of key histories.

The concept of building a "lightweight PKI" based on the normal operation of the network was also proposed in the context of securing BGP routing. A "Grassroots PKI" [12] binds a public key to a prefix of IP address space if that key is included in a stable and widely used routing announcement for that prefix.

10 Future Work

We believe that the network notary concept introduced by PERSPECTIVES opens several promising avenues for additional exploration in the area of host authentication.

Notary-Aware Services: As presented, PERSPECTIVES only requires client modifications. However, if notaries become common, servers might be modified to also communicate with notaries. This would provide three primary benefits:

1. **Immediate Probing of New Keys:** A server could immediately alert notaries when it comes online or changes its key, allowing notaries to quickly begin building an observation history for the new key.
2. **Reduced Need to Query Notaries:** The server could act as a caching proxy by querying notaries on behalf of clients. This would eliminate privacy issues related to clients querying notaries directly and would also allow clients to receive cached observed key data even if notaries were temporarily unavailable.
3. **Attack Detection:** With access to the notary infrastructure, a server could request observed key data for its own service-id and alert its administrator if the notary replies include any illegitimate keys. Such a false key is a likely indication that either a network element near the server is malicious or that a notary is compromised, thereby aiding attack detection.

Unlike standard PKIs, these changes do not require server owners to manually prove their identity to a third-party CA, making them simple to adopt.

Applying PERSPECTIVES to Additional Protocols: PERSPECTIVES opens the door for more widespread use of SSH-style authentication with other protocols because, unlike Tofu, use of key-trust policies can *automatically* authenticate keys in a secure fashion, even on the first connection to a service.

SMTP: Many SMTP servers already have self-signed SSL certificates so that local clients who manually install these certificates can authenticate their outgoing mail server. However, because no PKI exists for one server to verify another server's certificate, emails are often transmitted "in-the-clear," leaving them vulnerable to snooping by whomever controls the intermediate networks. PERSPECTIVES could support mutual authentication of inter-SMTP server communication, allowing, for example, a server to refuse to transmit a message the user has deemed "sensitive" to an unauthenticated server.¹⁵

Incremental DNSSEC: The deployment of secure DNS (DNSSEC) is hampered by the fact that a sub-domain (e.g.,

¹⁵DomainKeys [6] could also benefit from PERSPECTIVES, as keys are currently acquired using unauthenticated DNS look-ups.

example.com) cannot protect its hosts until its parent domain (e.g., .com) publishes its own public key and signs the sub-domain's public key. Unfortunately, to date, major top-level domains have shown little enthusiasm for deploying DNSSEC. With PERSPECTIVES, the authoritative nameserver for any sub-domain could publish an un-signed key used to sign its zone, with resolving name servers using notaries to validate key prior to caching.

11 Conclusion

As evidenced by its widespread use, SSH-style host authentication offers a simple and attractive alternative to a heavyweight PKI. Unfortunately, "Trust-on-first-use" leaves users vulnerable to simple MitM attacks, limiting the effectiveness of current Tofu applications and preventing other protocols from being able to take advantage of lightweight SSH-style host authentication. To enhance security without requiring a PKI, we designed PERSPECTIVES to supplement Tofu-based applications with spatial and temporal redundancy. Our implementation demonstrates that the notary concept is practical, and after using our PERSPECTIVES clients for nearly a year, we have found them invaluable at several occasions: when logging in to a new server while connecting through a public wireless network, or when connecting to a known server after a server key change. As a result, we believe that PERSPECTIVES is a practical approach to improving the security of users communicating with SSH and self-signed HTTPS.

Acknowledgments

Dan Wendlandt was supported by a graduate fellowship from the Dept. of Homeland Security and an award from the ARCS Foundation. This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, grants CCF-0424422 and CNS-0716278 from the National Science Foundation, and a Sloan Foundation faculty fellowship. The views contained here are those of the authors and do not necessarily represent the official policies or endorsements of ARO, CMU, NSF, or the U.S. Government. We thank Bryan Parno, Himabindu Pucha, and our many reviewers for useful comments. Special thanks to Ramu Panayappan for developing the Perspectives extension for Firefox.

References

- [1] Y. Ali and S. Smith. Flexible and scalable public key security for SSH. In *EuroPKI*, pages 43–56, 2004.
- [2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Experience with an Evolving Overlay Network Testbed. *ACM Computer Communications Review*, 33(3):13–19, July 2003.
- [3] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. *Lecture Notes in Computer Science*, 1666:431–448, 1999.

- [4] J. Callas, L. Donnerhake, H. Finney, D. Shaw, and R. Thayer. *OpenPGP Message Format*. Internet Engineering Task Force, Nov. 2007. RFC 4880.
- [5] S. Chokhani and W. Ford. *Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework*. Internet Engineering Task Force, 1999. RFC 2527.
- [6] M. Delany. *Domain-based Email Authentication Using Public Keys Advertised in the DNS (DomainKeys)*, Aug. 2004.
- [7] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, pages 107–125, 1992.
- [8] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proc. 13th USENIX Security Symposium*, Aug. 2004.
- [9] Drive-by Pharming. Symantec security response weblog: Drive-by pharming in the wild. http://www.symantec.com/enterprise/security_response/weblog/2008/01/driveby_pharming_in_the_wild.html.
- [10] S. Egelman, L. F. Cranor, and J. Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '08)*, 2008.
- [11] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral. In *Proceedings of the 4th USENIX Symposium on Network Systems Design and Implementation (NSDI)*, 2004.
- [12] Y.-C. Hu, D. McGrew, A. Perrig, B. Weis, and D. Wendlandt. (R)Evolutionary bootstrapping of a global PKI for securing BGP. In *Proc. 5th ACM Workshop on Hot Topics in Networks (Hotnets-V)*, Nov. 2006.
- [13] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Symposium on Network and Distributed Systems Security (NDSS '99)*, Feb. 1999.
- [14] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
- [15] S. Miller, B. Neuman, J. Schiller, and J. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, Oct. 1988. Project Athena Technical Plan.
- [16] NTP-Pool. pool.ntp.org: the Internet cluster of NTP servers. <http://www.pool.ntp.org>.
- [17] OpenSSH. <http://www.openssh.com>.
- [18] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. 1st ACM Workshop on Hot Topics in Networks (Hotnets-I)*, Oct. 2002.
- [19] L. Poole and V. S. Pai. ConfiDNS: Leveraging scale and history to improve DNS security. In *Proceedings of Third Workshop on Real, Large Distributed Systems (WORLDS)*, November 2006.
- [20] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. 2008.
- [21] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.
- [22] J. Schlyter and W. Griffin. *Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints*. Internet Engineering Task Force, Jan. 2006. RFC 4255.
- [23] The RON/IRIS Testbed. <http://www.datapostory.net/tb/>.
- [24] Tor Exit Node Hijacks. TOR exit-node doing MITM attacks. <http://www.teamfurry.com/wordpress/2007/11/20/tor-exit-node-doing-mitm-attacks>.
- [25] Traceroute.org. Traceroute.org. <http://www.traceroute.org>.
- [26] W32.Arpfame. W32.arpfame. http://http://www.symantec.com/business/security_response/writeup.jsp?docid=2007-061222-0609-99.

Spectator: Detection and Containment of JavaScript Worms

Benjamin Livshits

Weidong Cui

Microsoft Research

Abstract

Recent popularity of interactive AJAX-based Web 2.0 applications has given rise to a new breed of security threats: JavaScript worms. In this paper we propose Spectator, the first automatic detection and containment solution for JavaScript worms. Spectator performs distributed data tainting by observing and tagging the traffic between the browser and the Web application. When a piece of data propagates too far, a worm is reported. To prevent worm propagation, subsequent upload attempts performed by the same worm are blocked. Spectator is able to detect fast and slow moving, monomorphic and polymorphic worms with a low rate of false positives. In addition to our detection and containment solution, we propose a range of deployment models for Spectator, ranging from simple intranet-wide deployments to a scalable load-balancing scheme appropriate for large Web sites.

In this paper we demonstrate the effectiveness and efficiency of Spectator through both large-scale simulations as well as a case study that observes the behavior of a real-life JavaScript worm propagating across a social networking site. Based on our case study, we believe that Spectator is able to detect all JavaScript worms released to date while maintaining a low detection overhead for a range of workloads.

1 Introduction

Web applications have been a prime target for application-level security attacks for several years. A number of attack techniques, including SQL injections, cross-site scripting, path traversal, cross-site request forgery, HTTP splitting, etc. have emerged, and recent surveys have shown that the majority of Web sites in common use contain at least one Web application security vulnerability [38, 42]. In fact, in the last several years, Web application vulnerabilities have become significantly more common than vulnerabilities enabled by

unsafe programming languages such as buffer overruns and format string violations [39].

While Web application vulnerabilities have been around for some time and a range of solutions have been developed [15, 17, 20, 22, 24, 29, 44], the recent popularity of interactive AJAX-based Web 2.0 applications has given rise to a new and considerably more destructive breed of security threats: JavaScript worms [11, 13]. JavaScript worms are enabled by cross-site scripting vulnerabilities in Web applications. While cross-site scripting vulnerabilities have been a common problem in Web based-applications for some time, their threat is now significantly amplified with the advent of AJAX technology. AJAX allows HTTP requests to be issued by the browser on behalf of the user. It is no longer necessary to trick the user into clicking on a link, as the appropriate HTTP request to the server can just be manufactured by the worm at runtime. This functionality can and has been cleverly exploited by hackers to create self-propagating JavaScript malware.

1.1 The Samy Worm

The first and probably the most infamous JavaScript worm is the Samy worm released on MySpace.com, a social networking site in 2005 [35]. By exploiting a cross-site scripting vulnerability in the MySpace site, the worm added close to a million users to the worm author's "friends" list. According to MySpace site maintainers, the worm caused an explosion in the number of entries in the friends list across the site, eventually leading to resource exhaustion. Two days after the attack the site was still struggling to serve requests at a normal pace.

The Samy worm gets its name from the MySpace login of its creator. Initially, the malicious piece of JavaScript (referred to as the *payload*) was manually placed in Samy's own MySpace profile page, making it infected. Each round of subsequent worm propagation consists of the following two steps:

1. **Download:** A visitor downloads an infected profile and automatically executes the JavaScript payload. This adds Samy as the viewer's "friend" and also adds the text *but most of all, samy is my hero* to the viewer's profile. Normally, this series of steps would be done through GET and POST HTTP requests manually performed by the user by clicking on various links and buttons embedded in MySpace pages. In this case, all of these steps are done in the background without the viewer's knowledge.
2. **Propagation:** The payload is extracted from the contents of the profile being viewed and then added to the viewer's profile.

Note that the enabling characteristic of a JavaScript worm is the AJAX propagation step: unlike "old-style" Web applications, AJAX allows requests to the server to be done in the background without user's knowledge. Without AJAX, a worm such as Samy would be nearly impossible. Also observe that worm propagation happens among properly authenticated MySpace users because only authenticated users have the ability to save the payload in their profiles.

1.2 Overview of the Problem

While Samy is a relatively benign proof-of-concept worm, the impact of JavaScript worms is likely to grow in the future. There are some signs pointing to that already: another MySpace worm released in December 2006 steals user passwords by replacing links on user's profile site with spoofed HTML made to appear like login forms [6]. The stolen credentials were subsequently hijacked for the purpose of sending spam. Similarly, Yamanner, a recent Yahoo! Mail worm, propagated through the Webmail system affecting close to 200,000 users by sending emails with embedded JavaScript to everyone in the current user's address book [4]. Harvested emails were then transmitted to a remote server to be used for spamming. Additional information on eight JavaScript worms detected in the wild so far is summarized in our technical report [21]. Interested readers are also referred to original vulnerability reports [4–6, 26, 33–35].

The impact of JavaScript worms will likely increase if attackers shift their attention to sites such as ebay.com, epinions.com, buy.com, or amazon.com, all of which provide community features such as submitting product or retailer reviews. The financial impact of stolen credentials in such a case could be much greater than it was for MySpace, especially if vulnerability identification is done with the aid of cross-site scripting vulnerability cataloging sites such as xssed.com [30]. Today cross-site scripting vulnerabilities are routinely exploited to allow

the attacker to steal the credentials of a small group of users for financial gain. Self-propagating code amplifies this problem far beyond its current scale. It is therefore important to develop a detection scheme for JavaScript worms before they become commonplace.

A comprehensive detection solution for JavaScript worms presents a tough challenge. The server-side Web application has no way of distinguishing a benign HTTP request performed by a user from one that is performed by a worm using AJAX. An attractive alternative to server-side detection would be to have an entirely client-side solution. Similarly, however, the browser has no way of distinguishing the origin of a piece of JavaScript: benign JavaScript embedded in a page for reasons of functionality is treated the same way as the payload of a worm. Filtering solutions proposed so far that rely on worm signatures to stop their propagation [37] are ineffective when it comes to polymorphic or obfuscated payloads, which are easy to create in JavaScript; in fact many worms detected so far are indeed obfuscated. Moreover, overly strict filters may cause false positives, leading to user frustration if they are unable to access their own data on a popular Web site.

1.3 Paper Contributions

This paper describes Spectator, a system for detecting and containing JavaScript worms, and makes the following contributions:

- Spectator is the first practical solution to the problem of detecting and containment of JavaScript worms. Spectator is also insensitive to the worm propagation speed; it can deal with rapid zero-day worm attacks as well as worms that disguise their presence with slow propagation. Spectator is insensitive of what the JavaScript code looks like and does not rely on signatures of any sort; therefore it is able to detect polymorphic worms or worms that use other executable content such as VBScript or JavaScript embedded in Flash or other executable content.
- We propose a scalable detection solution that adds a small constant-time overhead to the end-to-end latency of an HTTP request no matter how many requests have been considered by Spectator. With this detection approach, Spectator is able to detect all worms that have been found in the wild thus far.
- Our low-overhead approximate detection algorithm is mostly conservative, meaning that for the majority of practical workloads it will not miss a worm if there is one, although false positives may be possible. However, simulations we have performed show that false positives are unlikely with our detection scheme.

- We propose multiple deployment models for Spectator: the Spectator proxy can be used as a server-side proxy or as a browser proxy running in front of a large client base such as a large Intranet site. For large services such as MySpace, we describe how Spectator can be deployed in a load-balanced setting. Load balancing enables Spectator to store historical data going far back without running out of space and also improves the Spectator throughput.
- We evaluate Spectator in several settings, including a large-scale simulation setup as well as a real-life case study using a JavaScript worm that we developed for a popular open-source social networking application deployed in a controlled environment.

1.4 Paper Organization

The rest of the paper is organized as follows. Section 2 describes the overall architecture of Spectator. We formally describe our worm detection algorithm and Spectator implementation in Sections 3 and 4, respectively. Section 5 describes the experiments and case studies we performed. Section 6 discusses Spectator design choices, tradeoffs, and threats to the validity of our approach. Finally, Sections 7 and 8 summarize related work and provide our conclusions.

2 Spectator Design Overview

This section provides an overview of Spectator architecture and design assumptions. Section 3 gives a formal description of our worm detection algorithm.

2.1 Spectator Overview

A recent study concluded that over 90% of Web applications are vulnerable to some form of security attack, including 80% vulnerable to cross-site scripting [42]. Cross-site scripting, which is at the root of JavaScript worms, is commonly identified as the most prevalent Web application vulnerability.

While it is widely recognized that secure programming is the best defense against application-level vulnerabilities, developing fully secure applications remains a difficult challenge in practice. For example, while MySpace was doing a pretty good job filtering well-formed JavaScript, it failed to filter out instances of `java\ascript`, which are interpreted as legal script in Internet Explorer and some versions of Safari. Despite best intentions, insecure applications inevitably get deployed on widely used Web sites.

The goal of Spectator is to protect Web site users from the adverse effects of worm propagation *after* the server

has failed to discover or patch a vulnerability in a timely manner. The essence of the Spectator approach is to *tag* or mark HTTP requests and responses so that copying of the content across a range of pages in a worm-like manner can be detected. Note that JavaScript worms are radically different from “regular” worms in that they are centralized: they typically affect a single Web site or a small group of sites (the same-origin policy of JavaScript makes it difficult to develop worms that propagate across multiple servers).

Spectator consists of an HTTP proxy inspecting the traffic between the user’s browser and a Web server in order to detect malicious patterns of JavaScript code propagation. Our tagging scheme described in Section 4 is a form of distributed tainting: whenever content that contains HTML is uploaded to the server, Spectator modifies it to attach a tag invisible to the end-user. The tag is preserved on the server and is contained in the HTML downloaded by subsequent requests. Spectator injects client-side support so that tags are reliably propagated on the client side and cannot be removed by worms aware of our tagging scheme. Client-side support relies on HTTP-only cookies and does not require specialized plug-ins or browser modifications, thus removing the barrier to client-side adoption.

Worm detection at the Spectator proxy works by looking for long propagation chains. Our detection algorithm is designed to scale to propagation graphs consisting of thousands of nodes with minimal overhead on every request. Whenever a long propagation chain is detected, Spectator disallows further uploads that are caused by that chain, thereby containing further worm propagation.

The Spectator detection algorithm is designed to detect propagation activity that affects multiple users. With every HTML upload, we also record the IP address of the user issuing the request. The IP address is used as an approximation of user identity. We keep track of IP addresses so that a user repeatedly updating their profile is not flagged as worm. If multiple users share an IP address, such as users within an intranet, this may cause false negatives. If the same user connects from different IP addresses, false positives might result. Worm detection relies on sufficiently many users adopting Spectator. However, since Spectator relies on no additional client-side support, it can be deployed almost instantaneously to a multitude of users.

2.2 Spectator Architecture

To make the discussion above more concrete, a diagram of Spectator’s architecture is shown in Figure 1. Whenever a user attempts to download a page containing Spectator tags previously injected there by Spectator, the following steps are taken, as shown in the figure:

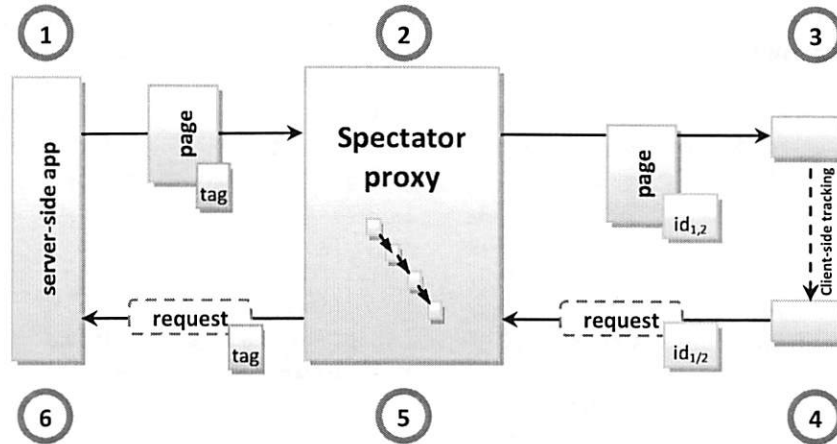


Figure 1: Spectator architecture

1. The tagged page is retrieved from the server.
2. The Spectator proxy examines the page. If the page contains tags, a new session ID is created and associated with the list of tags in the page. The tags are stripped from the page and are never seen by the browser or any malicious content executing therein.
3. The modified page augmented with the session ID stored in a cookie (referred to below as “Spectator cookie”) is passed to the browser.

Whenever an upload containing HTML is observed, the following steps are taken:

4. If a Spectator cookie is found on the client, it is automatically sent to Spectator by the browser (the cookie is the result of a previous download in step 3).
5. If the request has HTTP content, a new *tag*—a number uniquely identifying the upload—is created by the Spectator proxy. If the request has a valid session ID contained in a Spectator cookie attached to the request, the list of tags it corresponds to is looked up and, for every tag, an edge between the old and the new tags are added to the propagation graph to represent tag causality. The request is not propagated further if the detection algorithm decides that the request is part of worm propagation.
6. Finally, the request augmented with the newly created tag is uploaded and stored at the server.

The Spectator worm detection algorithm relies on the following properties that guarantee that we can observe and record the propagation of a piece of data during its entire “round trip”, captured by steps 1–6 above, thereby enabling taint tracking. The properties described below give the information required to formally reason about the Spectator algorithm. A detailed discussion of how

Spectator ensures that these properties hold is delayed until Section 4.

Property 1: Reliable HTML input detection. *We can detect user input that may contain HTML and mark it as tainted. Additionally, we can mark suspicious user input without disturbing server-side application logic so that the mark propagates to the user.*

Property 2: Reliable client-side tag propagation.

Browser can propagate taint tags from an HTTP response to a subsequent request issued by the browser.

3 Worm Detection Algorithm

This section describes our worm detection algorithm. Section 3.1 formalizes the notion of a worm and Section 3.2 talks about our detection algorithm. Finally, Section 3.3 discusses worm containment.

3.1 Propagation Graph Representation

We introduce the notion of a *propagation graph* that is updated whenever new tags are inserted. Each node of the graph corresponds to a tag and edges represent causality edges. Each node carries with it the IP address of the client the tag originates from.

Definition 1. A tag is a long integer uniquely identifying an HTML upload to Spectator.

Definition 2. A causality edge is a tuple of tag-IP address pairs $\langle (t_1, ip_1), (t_2, ip_2) \rangle$ representing the fact that t_2 requested by ip_2 originated from a page requested by ip_1 that has t_1 associated with it.

Definition 3. Propagation graph $G = \langle \mathcal{V}, \mathcal{E} \rangle$, where vertices \mathcal{V} is a set of tag-IP pairs

$\{(t_1, ip_1), (t_2, ip_2), \dots\}$ and \mathcal{E} is the set of causality edges between them.

Definition 4. The distance between two nodes N_1 and N_2 , denoted as $|N_1, N_2|$, in a propagation graph G is the smallest number of unique IP addresses on any path connecting N_1 and N_2 .

Definition 5. Diameter of a propagation graph G , denoted as $\mathcal{D}(G)$, is the maximum distance between any two nodes in G .

Definition 6. We say that G contains a worm if $\mathcal{D}(G)$ exceeds a user-provided threshold d .

Note that the propagation graph is acyclic. While it is possible to have node sharing, caused by a page with two tags generating a new one having a cycle in the propagation graph is impossible, as it would indicate a tag caused by another one that was created chronologically later. Ideally, we want to perform worm detection on the fly, whenever a new upload request is observed by Spectator. When a new edge is added to the propagation graph G , we check to see if the diameter of updated graph G now exceeds the user-defined threshold d .

The issue that complicates the design of an efficient algorithm is that we need to keep track of the set of unique IP addresses encountered on the current path from a root of the DAG. Unfortunately, computing this set every time an edge is added is exponential in the graph size in the worst case. Storing the smallest set of unique IP addresses at every node requires $O(n^2)$ space in the worst case: consider the case of a singly-linked list where every node has a different IP address. Even if we store these sets at every node, the computation of the IP address list at a node that has more than one predecessor still requires an exponential amount of work, as we need to consider all ways to traverse the graph to find the path with the smallest number of unique IP addresses. Our goal is to have a worm detection algorithm that is as efficient as possible. Since we want to be able to detect slow-propagating worms, we cannot afford to remove old tags from the propagation graph. Therefore, the algorithm has to scale to hundreds of thousands of nodes, representing tags inserted over a period of days or weeks.

3.2 Incremental Approximate Algorithm

In this section we describe an iterative algorithm for detecting when a newly added propagation graph edge indicates the propagation of a worm. As we will demonstrate later, the approximation algorithm is mostly conservative, meaning that if there is a worm, in most cases, the approximation approach will detect it *no later* than the precise one.

3.2.1 Data Representation

The graph G_A maintained by our algorithm is a forest approximating the propagation graph G . Whenever node sharing is introduced, one of the predecessors is removed to maintain the single-parent property. Furthermore, to make the insertion algorithm more efficient, some of the nodes of the graph are designated as *storage stations*; storage stations accelerate the insertion operation in practice by allowing to “hop” towards a root of the forest without visiting every node on the path.

We use the following representation for our approximate algorithm. $PREV(N)$ points to the nearest storage station on its path to the root or null if N is the root. Every node N in G_A has a set of IP addresses $IPS(N)$ associated with it. The number of IP addresses stored at a node is at most c , where c is a user-configured parameter. At every node N we maintain a depth value denoted as $DEPTH(N)$, which is an approximation of the number of unique IP addresses on the path from N to the root. Whenever the $DEPTH$ value exceeds the user-defined threshold d , we raise an alarm.

3.2.2 Worm Detection

For space reasons, detailed pseudo-code for the insertion algorithm that describes the details of data structure manipulation is given in our technical report [21]. Here we summarize the essence of the insertion algorithm. Whenever a new causality edge from node *parent* to node *child* is added to G_A :

1. If *parent* is the only predecessor of *child* in G_A , we walk up the tree branch and find all storage stations on the current tree branch. We copy $IPS(parent)$ into $IPS(child)$ and then add *child*’s IP if it is not found by the search. In the latter case, $DEPTH(child)$ value is incremented. If the size of $IPS(child)$ reaches threshold c , we designate *child* as a storage station.
2. If *child* has two predecessors in G_A , we compare $DEPTH$ values stored at the two predecessors, select the larger one, and remove the other edge from the graph, restoring non-sharing. After that we follow step 1 above. Note that the predecessors do not have to belong to the same tree. However, after the insertion is complete, *child* will be a member of a single tree.

Observe that the the *maximum DEPTH* value computed by this algorithm is exactly $\mathcal{D}(G_A)$ because the maximum distance in G_A is that between a node and a root.

Notice that the approach described in this section is essentially a greedy algorithm: in the presence of multiple parents, it chooses the parent that it believes will result in

$$IPS(N) = \begin{cases} \text{IP addresses on the path from } N \text{ to } PREV(N) \text{ not contained in any} & \text{if } PREV(N) \neq \text{null} \\ \text{other } IPS \text{ sets of nodes between } N \text{ and the root} & \\ \text{IP addresses on the path from } N \text{ to the root} & \text{if } PREV(N) = \text{null} \end{cases}$$

Figure 2: Definition of *IPS*.

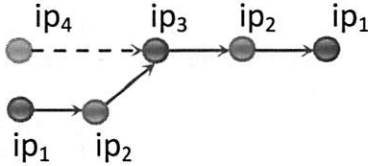


Figure 3: Propagation graph for which the approximate algorithm under-approximates the diameter value

higher overall diameter of the final approximation graph G_A . Of course, the advantage of the approximate algorithm is that it avoids the worst case exponential blow-up. However, without the benefit of knowing future insertion operations, the greedy algorithm may yield a lower diameter, potentially leading to false negatives. While this has never happened in our experiments, one such example is described below.

Example 1. Consider the propagation graph displayed in Figure 3. Suppose we first insert the two nodes on the bottom left with IPs ip_1 and ip_2 and then the node with ip_4 . When we add ip_3 to the graph, the approximation algorithm will decide to remove the newly created edge (showed as dashed) because doing so will result in a greater diameter. However, the greedy algorithm makes a *suboptimal* decision: when nodes on the right with IPs ip_2 and ip_1 are added, the resulting diameter will be 3, not 4 as it would be with the precise approach. \square

3.2.3 Incremental Algorithm Complexity

Maintaining an approximation allows us to obtain a very modest time and space bounds on new edge insertion, as shown below. Discussion of how our approximate detection algorithm performs in practice is postponed until Section 5.

Insertion Time Complexity. The complexity of the algorithm at every insertion is as follows: for a graph G_A with n nodes, we consider d/c storage stations at the most. Since storage stations having non-overlapping lists of IP addresses, having *more* storage stations on a path from a root of G_A would mean that we have over d IPs in total on that particular path, which should have been detected as a worm. At every storage station, we perform an $O(1)$ average time containment check. So, as a result, our approximate insertion algorithm takes $O(1)$ time on average.

Space Complexity. We store $O(n)$ IP addresses at the storage stations distributed throughout the propagation graph G_A . This is easy to see in the worst case of every IP address in the graph being unique. The union of all IP lists stored at all storage stations will be the set of all graph nodes. Additionally, we store IP addresses at the nodes *between* subsequent storage stations. In the worst case, every storage station contains c nodes and we store $1 + 2 + \dots + c - 1 = c \cdot (c - 1)/2$ IP addresses, which preserves the total space requirement of $O(n)$. More precisely, with at most n/c storage stations, we store approximately

$$\frac{c^2}{2} \times \frac{n}{c} = \frac{1}{2} \cdot c \cdot n$$

IP addresses. Note that in practice storage stations allow insertion operations to run faster because instead of visiting every node on the path from the root, we can instead “hop” to the next storage station, as demonstrated by the d/c bound. However, using storage stations also results in more storage space being taken up as shown by the $1/2 \cdot c \cdot n$ bound. Adjusting parameter c allows us to explore this space-time trade-off: bigger c results in faster insertion times, but also requires more storage.

Worm Containment Complexity. When a worm is detected, we walk the tree that the worm has infected and mark all of its nodes as such. This takes $O(n)$ time because in the worst case we have to visit and mark all nodes in the tree. The same bound holds for when we mark nodes in a tree as false positives.

3.3 Worm Containment

Whenever the depth of the newly added node exceeds detection threshold d , we mark the entire tree containing the new edge as infected. To do so, we maintain an additional status at every leaf. Whenever a tree is deemed infected by our algorithm, we propagate the infected status to every tree node. Subsequently, all uploads that are caused by nodes within that tree are disallowed until there is a message from the server saying that it is safe to do so.

When the server fixes the vulnerability that makes the worm possible, it needs to notify the Spectator proxy, at which point the proxy will remove the entire tree containing the new edge from the proxy. If the server deems the

vulnerability reported by Spectator to be a false positive, we never subsequently report activity caused by nodes in this tree as a worm. To do so, we set the node status for each tree node as a false positive and check the node status before reporting a worm.

4 Spectator Implementation

Distributed tainting in Spectator is accomplished by augmenting both upload requests to insert tracking tags and download requests to inject tracking cookies and JavaScript.

4.1 Tag Propagation in the Browser

To track content propagation on the client side, the Spectator proxy maintains a local *session* for every page that passes through it. Ideally, this functionality would be supported by the browser natively; in fact, if browsers supported *per-page cookies*, that is, cookies that expire once the current page is unloaded, this would be enough to precisely track causality on the client side. Since such cookies are not supported, we use a combination of standard per-session browser cookies and injected JavaScript that runs whenever the current page is unloaded to accomplish the same goal.

4.1.1 Client-Side Causality Tracking

Whenever a new page is sent by the Spectator proxy to the browser, a new *session tuple* $\langle id_1, id_2 \rangle$ is generated, consisting of two long integer values, which are randomized 128-bit integers, whose values cannot be easily guessed by the attacker. Our client-side support consists of two parts:

HTTP-only Spectator cookie in the browser. We augment every server response passing through Spectator with an HTTP-only cookie containing id_1 . The fact that the session ID is contained in an HTTP-only cookie means that it cannot be snooped on by malicious JavaScript running within the browser, assuming the browser correctly implements the HTTP-only attribute. For a page originating from server *D*, the domain of the session ID cookie is set to *D*, so it is passed back to Spectator on every request to *D*, allowing us to perform causality tracking as described above.

Ideally, we would like to have a per-page cookie that expires as soon as the page is unloaded. Unfortunately, there is no support for such cookies. So, we use session cookies that expire after the browser is closed, which may not happen for a while. So, if the user visits site *D* served by Spectator, then visits site *E*, and then returns to *D*, the Spectator cookie would still be sent to Spectator by the browser.

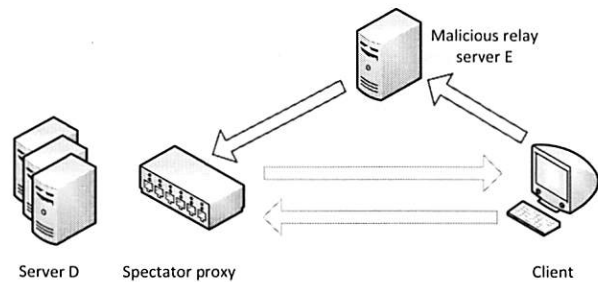


Figure 4: Relaying user requests through a malicious server

Injected client-side JavaScript to signal page unloads.

In order to terminate a propagation link that would be created between the two unrelated requests to server *D*, we inject client-side JavaScript into every file that Spectator sends to the browser. Furthermore, before passing the page to the client, within Spectator we add an unload event handler, which sends an XMLHttpRequest to the special URL `__spectator__` to “close” or invalidate the current session, so that subsequent requests with the same id_1 are ignored. The `__spectator__` URL does not exist on server *D*: it is just a way to communicate with Spectator while including id_1 created for server *D* (notice that it is not necessary to pass the session ID as a parameter to Spectator, as the session ID cookie will be included in the request as well).

Injected client-side code is shown in Figure 5. To make it so that malicious JavaScript code cannot remove the unload handler, we mediate access to functions `window.attachEvent` and `window.detachEvent` as suggested in the BEEP system [16] by injecting the JavaScript shown in Figure 6 at the very top of each page served by Spectator. Furthermore, we also store id_2 as a private member of class handler [8]; this way it is not part of the handler.unload function source code and cannot be accessed with a call to `toString`. To prevent id_2 from being accessed through DOM traversal, the original script blocks defining the unload handler and containing the numerical value of id_2 embedded verbatim is subsequently removed through a call to `removeChild` in the next script block, similar to what is suggested by Meschkat [25].

4.1.2 Attacks Against Client-Side Tracking

While the basic client-side support is relatively simple to implement, there are two types of potential attacks against our client-side scheme to address, as described below.

Worm Relaying. First, the attacker might attempt to break a propagation chain by *losing* the session ID contained in the browser, a technique we refer to as *worm*

```

<script id="remove-me">
  if (window.attachEvent) {
    var handler = function(id) {
      var id2 = id;
      this.unload = function() {
        var xhr = new XMLHttpRequest("MSXML2.XMLHTTP.3.0");
        xhr.open("POST", "http://www.D.com/___spectator_&" + id2, true);
        xhr.send(null); // send message to D just before unloading
      };
      // embed id_2 verbatim and create an unload handler
      window.attachEvent("unload", (new handler(<id_2>)).unload);
    };
  }
</script>

<script>
  // remove the previous script block from the DOM
  var script_block = document.getElementById("remove-me");
  script_block.parentNode().removeChild(script_block);
</script>

```

Figure 5: Intercepting page unload events in JavaScript

relaying. Suppose we have a page in the browser loaded from server *D*. The attacker may programmatically direct the browser to a different server *E*, which would in turn connect to *D*. Server *E* in this attack might be set-up solely for the sole purpose of relaying requests to server *D*, as shown in Figure 4. Notice that since the session ID cookie will *not* be sent to *E* and its value cannot be examined. We introduce a simple restriction to make the Spectator proxy redirect all accesses to *D* that do *not* contain a session ID cookie to the top-level *D* URL such as *www.D.com*. In fact, it is quite common to disallow access, especially programmatic access through AJAX RPC calls, to an inside URL of large site such as *www.yahoo.com* by clients that do not already have an established cookie. With this restriction, *E* will be unable to relay requests on behalf of the user.

Tampering with unload events. To make it so that malicious JavaScript code cannot remove the unload handler or trigger its own unload handler after Spectator's, we mediate access to function `window.detachEvent` by injecting the JavaScript shown in Figure 6 at the very top of each page served by Spectator. If malicious script attempts to send the unload event to Spectator prematurely in an effort to break the propagation chain, we will receive more than one unload event per session. When a sufficient number of duplicate unload events is seen, we raise an alarm for the server, requiring a manual inspection. It is still possible for an attacker to try to cause false positives by making sure that the unload event will *never be sent*. This can be accomplished by crashing the browser by exploring a browser bug or trying to exhaust browser resources by opening new windows. However, this behavior is sufficiently conspicuous to the end-user to prompt a security investigation and is thus not a good

```

<script>
window.attachEvent = function(sEvent, fpNotify) {
  if (sEvent == "unload") return;
  window.attachEvent(sEvent, fpNotify);
}

window.detachEvent = function(sEvent, fpNotify) {
  if (sEvent == "unload") return;
  window.detachEvent(sEvent, fpNotify);
}
</script>

```

Figure 6: Disallow adding or removing unload event handlers

candidate for inclusion within a worm.

Opening a new window. Note that opening a new window will not help an attacker break causality chains. If they try to perform a malicious upload before the unload event in the original window is sent to the proxy, Spectator will add a causality link for the upload. Fetching a new page with no tags before the malicious upload will not help an attacker evade Spectator because this clean page and the original page share the same HTTP-only cookie. As such, Spectator will think that the upload is caused by either of the sessions corresponding to that cookie. This is because Spectator's approximation approach selects the parent node with a larger depth when a node has multiple predecessors.

4.2 Tagging Upload Traffic and Server-Side Support for Spectator

The primary goal of server-side support is to embed Spectator tags into suspicious data uploaded to a protected Web server in a transparent and persistent manner so that (1) the tags will not interfere with the Web

server's application logic; and (2) the embedded tags will be propagated together with the data when the latter is requested from the Web server. To achieve these goals of transparency and persistence, we need to be able to reliably detect suspicious data to embed Spectator tags into uploaded input. Next, we discuss our solutions to the challenges of transparency and persistence that do *not* require any support on the part of the Web server.

Data uploads are suspicious if they may contain embedded JavaScript. However, for a cross-site scripting attack to be successful, this JavaScript is usually surrounded with some HTML tags. The basic idea of detecting suspicious data is to detect the *presence* of HTML-style content in the uploaded data. Of course, such uploads represent a minority in most applications, which means that Spectator only needs to tag and track a small portion of all requests. Spectator detects suspicious data by searching for opening matching pairs of HTML tags `<tag attribute1 = ... attribute2 = ...>` and `</tag>`. Since many servers may require the uploaded data to be URL- or HTML-encoded, Spectator also attempts to decode the uploaded data using these encodings before attempting the pattern-matching.

Spectator embeds a tag immediately preceding the first opening `>` for each matching pair of HTML tags. (Note that if the original data is URL encoded, Spectator will re-encode the tagged output as well.) To illustrate how tag insertion works, consider an HTTP request containing parameter

```
<div><b onclick="javascript:alert(...)">...</b></div>
```

This parameter will be transformed by Spectator into a request containing

```
<div spectator_tag=56><b
  onclick="javascript:alert(...)"
  spectator_tag=56>...</b>
</div>
```

We tested this scheme with several real-world web servers chosen from a cross-site scripting vulnerability listing site `xssed.com`. For vulnerable servers that reflect user input verbatim, this scheme works well as expected. Our further investigations into three popular Webmail sites, Hotmail, Yahoo Mail, and Gmail, have shown this scheme did not work because the Spectator tags were stripped by the Web servers. While this is difficult to ascertain, our hypothesis is that these sites use a whitelist of allowed HTML attributes.

To handle Web sites that may attempt to strip Spectator tags, we propose an alternative approach. In this new scheme, Spectator embeds tags directly into the actual content surrounded by HTML tags. For example `hello world...` will be transformed by Spectator into a request containing ` spectator_tag = 56hello world...` We tested this scheme with the three Webmail sites above and found that it works for all of them. However, there is a possibility that such tags may interfere with Web server's application logic. For example, if the length of the actual content is

explicitly specified in the data, this tagging scheme will affect data consistency. Unfortunately, while our approach to decode and augment the uploaded traffic works for the sites we have experimented with, in the worst case, the server may choose an entirely new way to encode uploaded parameters. In this case, properly identifying and tagging HTML uploads will require server-side cooperation.

5 Experimental Evaluation

An experimental evaluation of Spectator poses a formidable challenge. Since we do not have access to Web sites on which real-life worms have been released, worm outbreaks are virtually impossible to replicate. Even if we were able to capture a set of server access logs, we still need to be able to replay the user activity that caused them. Real-life access patterns leading to worm propagation are, however, hard to capture and replay. Therefore, our approach is to do a large-scale simulation as well as a small-scale real-world study.

Large-scale simulations. We created OurSpace, a simple Web application that conceptually mimics the functionality of MySpace and similar social networking sites on which worms have been detected, but without the complexity of real sites. OurSpace is able to load and store data associated with a particular user given that user's ID. For faster access, data associated with the user is stored in an in-memory database with which OurSpace communicates. With the help of OurSpace, we have experimented with various access patterns that we feel reflect access patterns of a real-life site under attack.

A real-life case study. It is difficult to experiment with real-life released worms as we discussed earlier. Ideally, we want to have the following features for our experimental setup: (1) a real-life popular widely-deployed Web application or a popular Web site; (2) a running JavaScript worm; (3) users running widely used browsers; and (4) multiple users observed over a period of time. To make sure that our ideas work well in a practical setting, we performed a series of experiments against Siteframe, an open-source content management system that supports blogging features [3]. On a high level, Siteframe is similar to MySpace: a user can post to his own blog, respond to other people's posts, add other users as friends, etc. We used Siteframe "Beaumont", version 5.0.0B2, build 538, because it allows HTML tags in blog posts and does not adequately filter uploaded content for JavaScript.

For our experiments, both OurSpace and Siteface were deployed on a Pentium 4 3.6 Ghz machine with 3 GB of memory machine running Windows XP with Apache 2.2 Web server installed. We ran the Spectator proxy as an application listening to HTTP traffic on a local host. The Spectator proxy is implemented on top of AjaxScope, a flexible HTML and JavaScript rewriting framework [18, 19]. The Spectator proxy implementation consists of 3,200 lines of C# code. For ease of deployment, we have configured our HTTP client to forward requests to the port the Spectator proxy listens on; the proxy subsequently forwards requests to the appropriate server, although other deployment strategies are possible, as discussed in Section 6.1.

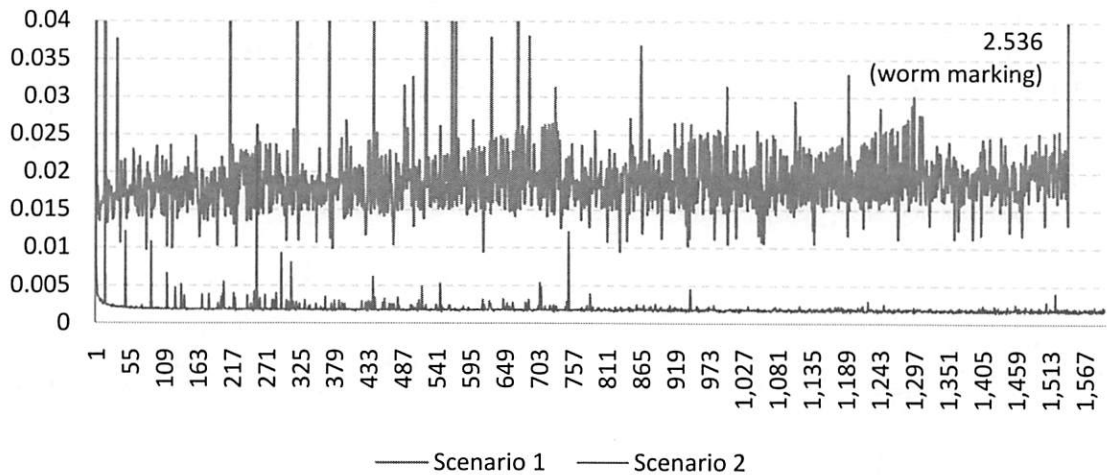


Figure 7: Propagation graph maintenance overhead, in ms, for Scenarios 1 (top) and 2 (bottom)

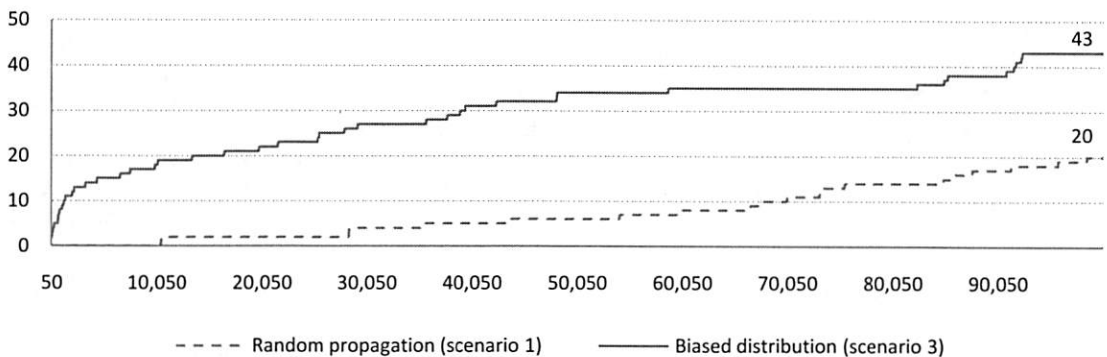


Figure 8: $D(G_A)$ values for random propagation (Scenario 1) vs biased distribution (Scenario 3)

5.1 OurSpace: Simulating User Behavior

In this section we describe access patterns that we believe to be representative of normal use.

Scenario 1: Worm outbreak (random topology). We have a pool of N users, each with a separate home page served by the Web application. Initially, user 1 wakes up and introduces malicious JavaScript into his profile page. At every step, a random user wakes up and visits a random page. If the visited page is infected, the user infects himself by embedding the content of the visited page into his home page. This simplified propagation model assumes worm traffic is the only HTML traffic that gets tagged. Regular non-HTML uploads do not lead to propagation edges being created. Note that this scenario can be easily augmented so that a user may view pages of multiple other users, thereby introducing sharing in the resulting propagation graph.

Scenario 2: A single long blog entry. We have a pool of N users that access the same blog page one after another. After user k accesses the page, he reads the previous $k - 1$ posts and then proceeds to create and uploads an HTML post that contains the previous posts and a new HTML post; the total diameter of the resulting graph is 2.

Scenario 3: A model of worm propagation (power law connectivity). To reflect the fact that some users are much more connected and active than others, in Scenario 3 we bias user selection towards users with a smaller ID using the power law. Most of the time, the user selection process heavily biases the selection towards users with a small ID. This bias reduces the set of users most actively participating in worm propagation, leading to “taller” trees being created.

5.2 Overhead and Scalability

To estimate the overhead, we experimented with Scenario 1 to determine how the approximate algorithm insertion time fluctuates as more nodes are added to the graph. Figure 7 shows insertion times for Scenario 1 with the detection threshold d set to 20. The x -axis corresponds to the tag being inserted; the y -axis shows the insertion time in milliseconds. The entire run took about 15 minutes with a total of 1,543 nodes inserted.

The most important observation about this graph is that the tag insertion latency is pretty much constant, hovering around .01 to .02 ms for Scenario 1 and close to .002 ms for Scenario 2. The insertion time for the second scenario is considerably lower, since the resulting approximation graph G_A

is much simpler: it contains a root directly connected to every other node and the maximum depth is 2. Since our proxy is implemented in C#, a language that uses garbage collection, there are few spikes in the graph due to garbage collection cycles. Also, the initial insertions take longer since the data structures are being established. Moreover, once the worm is detected at $d = 20$ for tag 1,543, there is another peak when all the nodes of the tree are traversed and marked.

5.3 Effectiveness of Detection

One observation that does not bode well for our detection approach with a random topology is that it takes a long time to reach a non-trivial depth. This is because the forest constructed with our approximation algorithm usually consists of set of shallow trees. It is highly unlikely to have a long narrow trace that would be detected as a worm before all the previous layers of the tree are filled up. However, we feel that the topology of worm propagation is hardly random. While researchers have tried to model worm propagation in the past, we are not aware of any work that models the propagation of JavaScript worms. We believe that JavaScript worms are similar to email worms in the way they spread. Propagation of JavaScript worms also tends to parallel social connections, which follow a set of well-studied patterns. Connectivity distribution is typically highly non-uniform, with a small set of popular users with a long tail of infrequent or defunct users. Similar observations have been made with respect to World Wide Web [1] and social network connectivity [2].

To properly assess the effectiveness of our approximation approach, we use Scenario 3, which we believe to be more representative of real-life topology. The simulation works as follows: initially, user 1's page is tainted with a piece of malicious JavaScript. At each step of the simulation, a user wakes up and chooses a page to view. The ID of the user to wake up and to view is chosen using the power law distribution. Viewing this page will create an edge in the propagation graph from the tag corresponding to the page selected for viewing to the newly created tag of the user that was awoken.

In propagation graphs generated using Scenario 3, once worm propagation reaches a well-connected node, it will tend to create much longer propagation chains involving that node and its friends. Figure 8 shows the diameter of G_A on the y -axis as more nodes are added up to 100,000 nodes for Scenarios 1 and 3, as shown on the x -axis. Observe that the diameter grows more rapidly in the case of selecting users from a biased distribution, as fewer nodes will be actively involved in propagation and shallow trees are less likely. This result indicates that in a real-life large-scale setting, which is likely to be similar to Scenario 3, our worm detection scheme is effective.

5.4 Precision of the Detection Algorithm

Note that as discussed in Section 3.3, the approximate algorithm detects the worm before the precise one in most cases. In fact, we have not encountered instances of when the approximate algorithm produces false negatives. However, a legitimate question is how much *earlier* is the worm detected with the approximate algorithm. If the approximate strategy is too eager

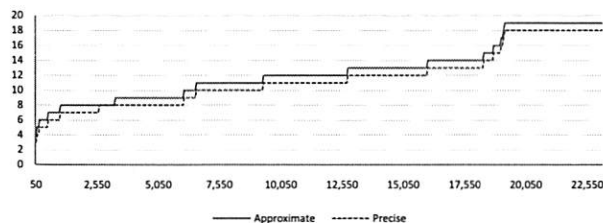


Figure 9: Approximate vs precise $D(G_A)$ values

to flag a worm, it will result in too many false positives to be practical. Whether that happens depends on the structure of the graph and the amount of sharing it has.

In order to gauge the detection speed obtained with the approximate scheme as opposed to the precise one we used simulations of Scenario 3 to generate a variety of random propagation graphs. We measured the diameter of the resulting propagation graph, as obtained from the precise and approximate methods. Figure 9 shows how $D(G)$ and $D(G_A)$ values, shown on the y -axis differ as more nodes are inserted, as shown on the x -axis for one such simulation. The differences between the two strategies are small, which means that we are not likely to suffer from false alarms caused by premature detection in practice, assuming a sufficiently high detection threshold. Furthermore, the approximation algorithm is always conservative in this simulation, over-approximating the diameter value.

5.5 Case Study: Siteframe Worm

For our experiments we have developed a proof-of-concept worm that propagates across a locally installed Siteframe site (The entire code of the Siteframe worm is presented in our technical report [21]). Conceptually our worm is very similar to how the Adulthood worm [32] works: the JavaScript payload is stored on an external server. At each propagation step, a new blog page is created, with a link to the worm payload embedded in it. This allows the worm to load the payload from the server repeatedly on every access. Whenever somebody visits the page, the worm executes and proceeds to create a new entry on the viewer's own blog that contains a link to the payload. To make our experiment a little easier to control, infection is triggered by the user clicking on an HTML <DIV> element. In real-life infection would probably occur on every page load.

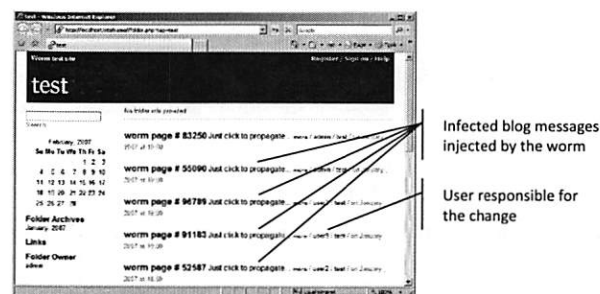


Figure 10: The main Siteframe site page after worm propagation

The worm does not check if a particular user has already been infected.

For our case study we created a total of five users on a fresh Siteframe site. Each user performed various activity on the site, leading to one or more worm propagation steps. The diameter of the resulting propagation graph was 5. To give a sense of the effects of worm propagation in this case, a screen shot of the resulting top-level page of the Siteframe site is shown in Figure 10. While small in scale, the Siteframe worm experiment has *significantly enhanced* our trust in the effectiveness of Spectator. Because the Siteframe worm was modeled after worms previously released in the wild, we believe that Spectator would have detected those worms.

6 Discussion

This section is organized as follows. Section 6.1 presents different deployment models for Spectator and Section 6.2 addresses threats to the validity of our approach.

6.1 Deployment Models for Spectator

Spectator works in both small-scale environments with servers that do not have a lot of activity and also with servers that have thousands of active users. We envision the following deployment models.

Server-side Deployment. Server-side deployment is the easiest way to protect an existing Web site from JavaScript worms using Spectator. Deploying the Spectator proxy in front of the server or servers that the site resides on allows the proxy to monitor all client-server interaction for that site and detect worms faster than it would in the case of being deployed elsewhere on the network and seeing only a portion of the total browser-server traffic. This model has the advantage of simplifying worm reporting, since the server is responsible for Spectator proxy maintenance.

Intranet-wide Deployment. Intranet deployment can be used to protect users within an organization, such as a university or a small enterprise against worm outbreaks. In many cases, these environments are already protected by firewalls and the Spectator proxy can be easily integrated within that infrastructure. Of course, worm detection in this kind of deployment is only possible if sufficiently many intranet users get infected. However, in the case of intranet deployment, the same proxy can be used to prevent worms propagating on a wide variety of sites without changes to our detection or tagging approaches.

A technical issue with client-side deployment is the use of SSL connections, which are not handled by the Spectator proxy. However, SSL sessions are frequently only used for initial authentication in Web applications and it is easy to set up one's browser to redirect requests to the Spectator proxy for non-SSL connections only. For server-side deployment though, the proxy can be placed before the SSL connection.

Large-scale Deployment. For large-scale server-side deployment, we may implement Spectator as part of the site's load balancer. Load balancing is a strategy used by most large-scale services such as MySpace or Live Spaces. When dealing with multiple proxies, our approach is to distribute different trees

in the forest G_A across the different proxy servers. The load balancer considers the source node of the edge being added to decide which proxy to redirect the request to. To avoid maintaining explicit state at the load-balancer, such as a lookup map that maps the parent tag to the proxy server containing that tree, our strategy is to assign the tag number *after* node insertion, based on which proxy it goes into. For instance, the last 5 bits of the tag may encode the number of the proxy to pass the request to. In the case of a node having more than one parent, we choose between two parents, based on the parent's depth as described in Section 3. When a proxy server is full and a new edge, whose parent resides on that proxy server is inserted, we migrate the newly inserted node to a different proxy server as a new tree. However, instead of the initial depth of 1, the depth of the root node for that tree is computed through our standard computation strategy.

While this deployment strategy closely matches the setup of large sites, an added advantage is the fact that we no longer have to store the entire forest in memory of a single proxy. A similar distributed strategy may be adopted for intranet-wide client-side deployment. Distributed deployment has the following important benefit: an attacker might try to avoid detection by flooding Spectator with HTML uploads, leading to memory exhaustion, and then unleashing a worm. Distributed deployment prevents this possibility.

6.2 Threats to Validity

The first and foremost concern for us when designing Spectator was limiting the number of false positives, while not introducing any false negatives. Recall that we require reliable HTML input detection and marking (see Property 1 in Section 2.2). Violations of this required property will foil our attempt to tag uploaded HTML and track its propagation, resulting in false negatives. However, Property 1 holds for all worms detected in the wild so far, as described in our technical report [21], and we believe that Spectator would have successfully detected them all. Still, potential for false positives remains, although without long-term studies involving large-scale data collection it is hard to say whether false positives will actually be reported in practice. Furthermore, it is possible for a group of benign users to perform the same activity a worm would run automatically. With a low detection threshold, the following manual worm-like activity is likely to be regarded as worm outbreaks.

Chain email in HTML format. As long as forwarding preserves HTML formatting, including Spectator tags, this activity will be flagged as a worm. Spectator has difficulty distinguishing this manual process from an automatic one such as the propagation of the Yamanner worm [4].

A long blog post. Similarly to a piece of chain mail, while a precise detection algorithm will not flag an excessively long blog post as a worm, the approximate algorithm will.

To avoid false positives, site administrators can set the detection thresholds higher. For instance, 500 is a reasonable detection threshold for Webmail systems and 1,000 is very conservative for blogging sites. As always, there is a trade-off between the possibility of false positives and the promptness of real worm

detection. However, a worm aware of our detection threshold may attempt to stop its propagation short of it [23].

7 Related Work

While to the best of our knowledge there has not been a solution proposed to JavaScript worms, there are several related areas of security research as described below.

7.1 Worm Detection

Since 2001, Internet worm outbreaks have caused severe damage that affected tens of millions of individuals and hundreds of thousands of organizations. This prompted much research on detecting and containing worms. However, most of the effort thus far has focused on worms that exploit vulnerabilities caused by unsafe programming languages, such as buffer overruns. Many techniques have been developed, including honeypots [9, 14, 41], dynamic analysis [7, 28], network traffic analysis [27, 36, 43], and worm propagation behavior [10, 45]. Our work is primarily related to research in the latter category. Xiong [45] proposes an attachment chain tracing scheme that detects email worm propagation by identifying the existence of transmission chains in the network. The requirement for monitoring multiple email servers limits the practicality of this scheme. Spectator, on the other hand, can observe all relevant traffic if deployed on the server side.

Ellis *et al.* [10] propose to detect unknown worms by recognizing uncommon worm-like behavior, including (1) sending similar data from one machine to the next, (2) tree-like propagation and reconnaissance, and (3) changing a server into a client. However, it is unclear how the behavioral approach can be deployed in practice because it is difficult to collect necessary information. We use a very basic worm-like behavior — long propagation chains — to detect JavaScript worms. Unlike Internet worms, JavaScript worms usually propagate inside the same Web domain. Spectator proposes an effective approach to achieve centralized monitoring, enabling worm detection. Our approach that only counts *unique* IP addresses on a propagation path is similar to looking at the propagation tree breadth in addition to its depth.

7.2 Server-side XSS Protection

There has been much interest in static and runtime protection techniques to improve the security posture of Web applications. Static analysis allows the developer to avoid issues such as cross-site scripting before the application goes into production. Runtime analysis allows exploit prevention and recovery. The WebSSARI project pioneered this line of research [15]. WebSSARI uses combined unsound static and dynamic analysis in the context of analyzing PHP programs. WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code. Several projects that came after WebSSARI improve on the quality of static analysis for PHP [17, 44]. The Griffin project proposes a scalable and precise sound static and runtime analysis techniques for finding security vulnerabilities in large Java applications [22,

24]. Based on a vulnerability description, both a static checker and a runtime instrumentation is generated. Static analysis is also used to drastically reduce the runtime overhead in most cases. The runtime system allows vulnerability recovery by applying user-provided sanitizers on execution paths that lack them. Several other runtime systems for taint tracking have been proposed as well, including Haldar *et al.* for Java [12] and Pietraszek *et al.* [31] and Nguyen-Tuong *et al.* for PHP [29].

7.3 Client-side Vulnerability Prevention

Noxes, a browser-based security extension, is designed to protect against information leakage from the user's environment while requiring minimal user interaction and customization effort [20]. Information leakage is a frequent side-effect of cross-site scripting attacks; e.g., the act of sending a cookie to an unknown URL will be detected and the user will be prompted. While effective at blocking regular cross-site scripting attacks, Noxes is generally helpless when it comes to data that is transmitted to a presumably trusted side without user's knowledge, such as it would be in the case of a JavaScript worm. In [40], Vogt *et al.* propose to prevent XSS on the client side by tracking the flow of sensitive information inside the web browser using dynamic data flow and static analysis. The main issues with their solution are the number of false alarms and how an average user can decide if an alarm is false.

8 Conclusions

This paper presents Spectator, the first practical detection and containment solution for JavaScript worms. The essence of the Spectator approach is to observe and examine the traffic between a Web application and its users, looking for worm-like long propagation chains. We have implemented and evaluated the Spectator solution on a number of large-scale simulations and also performed a case study involving a real JavaScript worm propagating across a social networking site. Our experiments confirm that Spectator is an effective and scalable, low-overhead worm detection solution.

Acknowledgments

We would like to express our profound gratitude to Karl Chen, Emre Kiciman, David Molnar, Berend-Jan "SkyLined" Wever, and others for their help in refining the ideas contained here and last-minute proof-reading help. Special thanks go to Úlfar Erlingsson and Martin Johns for their suggestions on how to implement client-side support. We are also grateful to the anonymous reviewers as well as our shepherd Sam King.

References

- [1] L. A. Adamic, B. A. Huberman, A. Barabási, R. Albert, H. Jeong, and G. Bianconi. Power-law distribution of the world wide web. *Science*, 287(5461):2115a+, Mar. 2000.
- [2] R. L. Breiger. *Dynamic Social Network Modeling and Analysis*. National Academies Press, 2004.
- [3] G. Campbell. Siteframe: a lightweight content-management system. <http://siteframe.org/>.

- [4] E. Chien. Malicious Yahoo!ligans. <http://www.symantec.com/avcenter/reference/malicious.yahoo!ligans.pdf>, Aug. 2006.
- [5] S. Corporation. Acts.spaceflash. http://www.symantec.com/security_response/writeup.jsp?docid=2006-071811-3819-99&tabid=2, July 2006.
- [6] S. Corporation. JS.Qspace worm. http://www.symantec.com/enterprise/security_response/writeup.jsp?docid=2006-120313-2523-99&tabid=2, Dec. 2006.
- [7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of Symposium on the Operating Systems Principles*, Oct. 2005.
- [8] D. Crockford. Private members in JavaScript. <http://www.crockford.com/javascript/private.html>, 2001.
- [9] W. Cui, V. Paxson, and N. Weaver. GQ: realizing a system to catch worms in a quarter million places. Technical Report TR-06-004, ICSI, Sept. 2006.
- [10] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia. A behavioral approach to worm detection. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, October 2004.
- [11] J. Grossman. Cross-site scripting worms and viruses: the impending threat and the best defense. <http://www.whitehatsec.com/downloads/WHXSSThreats.pdf>, Apr. 2006.
- [12] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Dec. 2005.
- [13] B. Hoffman. Analysis of Web application worms and viruses. <http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Hoffman/BH-Fed-06-Hoffman-up.pdf>, 2006.
- [14] Honeynet. The honeynet project. <http://www.honeynet.org/>.
- [15] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the Conference on World Wide Web*, pages 40–52, May 2004.
- [16] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-enforced embedded policies. Technical report, Department of Computer Science, University of Maryland, 2006.
- [17] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities (short paper). In *Proceedings of the Symposium on Security and Privacy*, May 2006.
- [18] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.
- [19] E. Kiciman and H. J. Wang. Live monitoring: using adaptive instrumentation and analysis to debug and maintain Web applications. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.
- [20] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the Symposium on Applied Computing*, Apr. 2006.
- [21] B. Livshits and W. Cui. Spectator: Detection and containment of JavaScript worms. Technical Report MSR-TR-2007-55, Microsoft Research, 2007.
- [22] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Aug. 2005.
- [23] J. Ma, G. M. Voelker, and S. Savage. Self-stopping worms. In *Proceedings of the ACM Workshop on Rapid malcode*, pages 12–21, 2005.
- [24] M. Martin, B. Livshits, and M. S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, Oct. 2006.
- [25] S. Meschkat. JSON RPC: Cross site scripting and client side Web services. In *23rd Chaos Communication Congress*, Dec. 2006.
- [26] M. Murphy. Xanga hit by script worm. <http://blogs.securiteam.com/index.php/archives/166>, Dec. 2005.
- [27] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.
- [28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of Network and Distributed System Security Symposium*, February 2005.
- [29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, June 2005.
- [30] P. Petkov. The generic XSS worm. <http://www.gnucitizen.org/blog/the-generic-xss-worm>, June 2007.
- [31] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the Recent Advances in Intrusion Detection*, Sept. 2005.
- [32] RSnake. Adultspace XSS worm. <http://ha.ckers.org/blog/20061214/adultspace-xss-worm/>, Dec. 2006.
- [33] RSnake. Semi reflective XSS worm hits Gaiaonline.com. <http://ha.ckers.org/blog/20070104/semi-reflective-xss-worm-hits-gaiaonlinecom/>, Jan. 2007.
- [34] RSnake. U-dominion.com XSS worm. <http://ha.ckers.org/blog/20061214/adultspace-xss-worm/>, Jan. 2007.
- [35] Samy. The Samy worm. <http://namb.la/popular/>, Oct. 2005.
- [36] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.
- [37] SonicWALL. SonicWALL deploys protection against MySpace worm. <http://sonic-wall.blogspot.com/2006/12/sonicwall-deploys-protection-against.html>, Dec. 2006.
- [38] M. Surf and A. Shulman. How safe is it out there? <http://www.imperva.com/download.asp?id=23>, 2004.
- [39] Symantec Corporation. Symantec Internet security threat report, volume X, Sept. 2006.
- [40] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, and C. Kruegel. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium*, pages 1–2, 2007.
- [41] M. Vrabie, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.
- [42] WebCohort, Inc. Only 10% of Web applications are secured against common hacking techniques. <http://www.imperva.com/company/news/2004-feb-02.html>, 2004.
- [43] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. Technical Report HPL-2002-172, HP Labs Bristol, 2002.
- [44] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Aug. 2006.
- [45] J. Xiong. ACT: Attachment chain tracing scheme for email virus detection and control. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, Oct. 2004.

A Compacting Real-Time Memory Management System

Silviu S. Craciunas

Christoph M. Kirsch

Hannes Payer

Ana Sokolova

Horst Stadler

Robert Staudinger

*Department of Computer Sciences
University of Salzburg, Austria*

firstname.lastname@cs.uni-salzburg.at

Abstract

We propose a real real-time memory management system called Compact-fit that offers both time and space predictability. Compact-fit is a compacting memory management system for allocating, deallocating, and accessing memory in real time. The system provides predictable memory fragmentation and response times that are constant or linear in the size of the request, independently of the global memory state. We present two Compact-fit implementations and compare them to established memory management systems, which all fail to provide predictable memory fragmentation. The experiments confirm our theoretical complexity bounds and demonstrate competitive performance. In addition, we can control the performance versus fragmentation trade-off via our concept of partial compaction. The system can be parameterized with the needed level of compaction, improving the performance while keeping memory fragmentation predictable.

1 Introduction

We present a compacting real-time memory management system called Compact-fit (CF) together with a moving and a non-moving implementation. Compact-fit is an explicit memory management system for allocating, deallocating, and accessing memory objects. Memory fragmentation in CF is bounded by a compile-time parameter. In CF compaction may only happen upon freeing a memory object and involves moving a single memory object of the same size.

Memory in CF is partitioned into 16KB pages. Each page is an instance of a so-called size-class, which partitions a page further into same-sized page-blocks. We adapt the concept of pages and size-classes from [2]. A memory object is always allocated in a page of the smallest-size size-class whose page-blocks still fit the object. Memory objects larger than 16KB are currently

not supported. However, in a future version, arraylets [3] may be used to handle objects of larger size with CF's complexity bounds.

The key idea in CF is to keep the memory *size-class-compact* at all times. In other words, at most one page of each size-class may be not-full at any time while all other pages of the size-class are always kept full. Whenever a memory object is freed, a memory object in the not-full page is moved to take its place and thus maintain the invariant. If the not-full page becomes empty, it can be reused in any size-class. Using several list and bitmap data structures, free space can be found in constant time, upon an allocation request.

The moving CF implementation maps page-blocks directly to physically contiguous pieces of memory, and therefore requires moving memory objects for compaction. Allocation takes constant time in the moving implementation, whereas deallocation takes linear time if compaction occurs. Dereferencing requires an additional pointer indirection and takes constant time.

The non-moving CF implementation uses a block table (effectively creating a virtual memory) to map page-blocks into physical block-frames that can be located anywhere in memory. In this case, compaction merely requires re-programming the block table rather than moving memory objects. However, although compaction may be faster, deallocation still takes linear time in the size of the object due to the block table administration. For the same reason allocation also takes linear time in the non-moving implementation. Our experiments show that deallocation is faster in the non-moving implementation for configurations in which block-frames are at least 80B. Dereferencing requires two additional pointer indirection and takes constant time.

A pointer in CF is an address and an offset. The CF system therefore supports offset-based rather than address-based pointer arithmetics, which we elaborate on later in the paper. Note that, in principle, the moving implementation may also support address-based pointer

arithmetics since each memory object is allocated in a single physically contiguous piece, that may however move during compaction.

In both implementations we can relax the always-compact-requirement allowing for more than one not-full page per size-class. As a result deallocation takes less time: it reduces up to constant time. This way we formalize, control, and implement the trade-off between temporal performance and memory fragmentation.

We present the results of benchmarking both implementations, as well as implementations of non-compacting real-time memory management systems (Half-fit [11] and TLSF [10]) and traditional (non-real-time) memory management systems (First-fit [7], Best-fit [7], and Doug Lea's allocator [8]) using synthetic workloads.

The contributions of this paper are: the CF system, the moving and non-moving implementations, and the experimental results on bare metal and Linux.

The rest of the paper is organized as follows: Section 2 discusses the principles behind the design of the compacting real-time memory management system. The implementation details and the complexity issues are presented in Section 3. We discuss related work in Section 4, and present the experiments, results and comparisons in Section 5. Section 6 wraps up with discussion and conclusion.

2 Principles of Compact-Fit

We start by introducing the goals of memory management in general and the requirements for real-time performance in particular. Having the basis set, we present our proposal for a compacting memory management system that meets the real-time requirements. We focus on the conceptual issues in this section and on the technical details in the following section.

2.1 Basics of Memory Management

By memory management we mean dynamic memory management. Applications use the dynamic memory management system to allocate and free (deallocate) memory blocks of arbitrary size in arbitrary order. In addition, applications can use the memory management system for accessing already allocated memory, *dereferencing*. Memory deallocation can lead to memory holes, which can not be reused for future allocation requests, if they are too small: this is the fragmentation problem. The complexity of allocation and deallocation depends on the fragmentation. A way to fight fragmentation is by performing *compaction* or *defragmentation*: a process of rearranging the used memory space so that larger contiguous pieces of memory become available.

There are two types of dynamic memory management systems:

- *explicit*, in which an application has to explicitly invoke the corresponding procedures of the dynamic memory management system for allocating and deallocating memory, and
- *implicit*, in which memory deallocation is no longer explicit, i.e., allocated memory that is not used anymore is detected and freed automatically. Such systems are called garbage collectors.

In this paper we propose an explicit dynamic memory management system.

2.2 Real-Time Requirements

Traditional dynamic memory management strategies are typically non-deterministic and have thus been avoided in the real-time domain. The memory used by real-time programs is usually allocated statically, which used to be a sufficient solution in many real-time applications. Nowadays increasing complexity demands greater flexibility of memory allocation, so there is a need of designing dynamic real-time memory management systems. Even in soft real-time systems and general purpose operating systems there exist time-critical components such as device drivers that operate on limited amount of memory because of resource and/or security constraints and may require predictable memory management.

In an ideal dynamic real-time memory management system each unit operation (memory allocation, deallocation, and dereferencing) takes constant time. We refer to this time as the response time of the operation of the memory management. If constant response times can not be achieved, then bounded response times are also acceptable. However, the response times have to be predictable, i.e., bounded by the size of the actual request and not by the global state of the memory.

More precisely, real-time systems should exhibit *predictability* of response times and of available resources.

The fragmentation problem affects the predictability of the response times. For example, if moving objects in order to create enough contiguous space is done upon an allocation request, then the response time of allocation may depend on the global state of the memory.

Predictability of available memory means that the number of the actual allocations together with their sizes determines how many more allocations of a given size will succeed before running out of memory, independently of the allocation and deallocation history. In a predictable system also the amount of fragmentation is predictable and depends only on the actual allocated objects. In addition to predictability, fragmentation should

be minimized for better utilization of the available memory.

Most of the established explicit dynamic memory management systems [8, 7] are optimized to offer excellent best-case and average-case response times, but in the worst-case they are unbounded, i.e., they depend on the global state of the memory. Hence these systems do not meet the above mentioned requirement on predictability of response times.

Moreover, to the best of our knowledge, none of the established explicit dynamic memory management systems meets the memory predictability requirement since fragmentation depends on the allocation and deallocation history.

The memory management system that we propose offers bounded response times (constant or linear in the size of the request) and predictable memory fragmentation, which is achieved via compaction.

Performing compaction operations could be done in either *event-* or *time-triggered* manner. As the names suggest, event-triggered compaction is initiated upon the occurrence of a significant event, whereas time-triggered compaction happens at predetermined points in time. Our compaction algorithm is event-triggered, compaction may be invoked upon deallocation.

2.3 Abstract and Concrete Address Space

We now describe the compacting real-time memory management system. At first, we focus on the management of memory addresses. Conceptually, there are two memory layers: the *abstract address space* and the *concrete address space*. Allocated objects are placed in contiguous portions of the concrete address space. For each allocated object, there is exactly one abstract address in the abstract address space. No direct references from applications to the concrete address space are possible: an application references the abstract address of an object, which furthermore uniquely determines the object in the concrete space. Therefore the applications and the memory objects (in the concrete space) are decoupled. All memory operations operate on abstract addresses. We start by defining the needed notions and notations.

The abstract address space is a finite set of integers denoted by \mathbb{A} . An abstract address a is an element of the abstract address space, $a \in \mathbb{A}$.

The concrete address space is a finite interval of integers denoted by \mathbb{C} . Note that, since it is an interval, \mathbb{C} is contiguous. Moreover, both the concrete and the abstract address spaces are linearly ordered by the standard ordering of the integers. A concrete address c is an element of the concrete address space, $c \in \mathbb{C}$.

A memory object m is a subinterval of the concrete address space, $m \in \mathcal{I}(\mathbb{C})$. For each memory object, two

concrete addresses $c_1, c_2 \in \mathbb{C}$, such that $c_1 \leq c_2$, define its range, i.e., we have $m = [c_1, c_2] = \{x \mid c_1 \leq x \leq c_2\}$.

As mentioned above, a used abstract address refers to a unique range of concrete addresses, which represents a memory object. Vice versa, the concrete addresses of an allocated memory object are assigned to a unique abstract address. To express this formally, we define a partial map that assigns to an abstract address the interval of concrete addresses that it refers to. The abstract address partial map

$$\text{address} : \mathbb{A} \hookrightarrow \mathcal{I}(\mathbb{C})$$

maps abstract addresses to memory objects. We say that an abstract address a is in use if $\text{address}(a)$ is defined. The abstract address map is injective, i.e., different abstract addresses are mapped to different subintervals, and moreover for all abstract addresses $a_1, a_2 \in \mathbb{A}$ that are in use, if $a_1 \neq a_2$, then $\text{address}(a_1) \cap \text{address}(a_2) = \emptyset$.

Accessing a specific element in the concrete address space \mathbb{C} requires two pieces of information: the abstract address a and an offset o , pointing out which element in the memory object $m = \text{address}(a)$ is desired. Therefore the next definition: An abstract pointer denoted by a_p is a pair $a_p = (a, o)$, where a is an abstract address in use (!) and o is an offset, $o \in \{0, \dots, |\text{address}(a)| - 1\}$. By $|\cdot|$ we denote the cardinality of a set. The abstract pointer space is the set of all abstract pointers a_p , and it is denoted by \mathbb{A}_p . There is a one-to-one correspondence between \mathbb{A}_p and the allocated subset of \mathbb{C} . Each abstract pointer a_p refers to a unique concrete address c via the abstract pointer mapping

$$\text{pointer} : \mathbb{A}_p \rightarrow \mathbb{C}.$$

It maps an abstract pointer $a_p = (a, o)$ to the concrete address of the memory object $m = \text{address}(a)$ that is at position o with respect to the order on $\text{address}(a)$.

Let $\mathbb{A} = \{1, 2, 3\}$ and $\mathbb{C} = \{1, 2, \dots, 10\}$. Assume that three memory objects of different size are allocated: $\text{address}(1) = [2, 3]$, $\text{address}(2) = [6, 7]$ and $\text{address}(3) = [8, 10]$. The abstract addresses together with their offsets create abstract pointers, which are mapped to \mathbb{C} . For example, $\text{pointer}(1, 1) = 3$ and $\text{pointer}(3, 1) = 9$. Figure 1 depicts this situation.

We now elaborate the benefits of using an abstract memory space. All references are redirected via the abstract memory space. An application points to a concrete memory location via an abstract pointer, cf. Figure 2(a).

Large data-structures often consist of a number of allocated memory objects connected via references (e.g. linked lists or trees) that depict the dependencies between the objects. These dependencies are handled via abstract pointers as well. This situation is also shown in Figure 2(a).

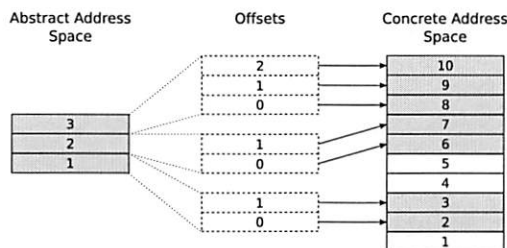


Figure 1: Abstract address and pointer mapping

Indirect referencing facilitates predictability of reference updates during compaction. If fragmentation occurs, the concrete address space \mathbb{C} gets compacted and the references from the abstract address space \mathbb{A} to the concrete address space \mathbb{C} are updated, as shown in Figure 2. Hence, objects are moved in \mathbb{C} and references are updated in \mathbb{A} . The number of reference updates is bounded: movement of one memory object in \mathbb{C} leads to exactly one reference update in \mathbb{A} . In contrast, direct referencing (related to object dependencies) would imply unpredictable number of reference updates. This is why we chose for an abstract address space design. However, note that the abstract address space is only required for predictable reference updating during compaction but otherwise completely orthogonal to the compaction algorithm and its moving and non-moving implementations described below.

The CF system provides three explicit memory operations (whose implementation we discuss in Section 3):

- `malloc(size)` is used to create a memory object of a given size. It takes an integer `size > 0` as argument and returns an abstract pointer $a_p = (a, o)$, where a is an abstract address that references to the allocated memory object and the offset o is set to 0, the beginning of the memory object.
- `free(a)` takes an abstract address a as argument and frees the memory object that belongs to this abstract address. The abstract address mapping is released.
- `dereference(a_p)` returns the concrete address c of an abstract pointer $a_p = (a, o)$, where a is the abstract address of a memory object and the offset o points to the actual position within the memory object.

Note that the abstract address of an allocated memory object never changes until the object gets freed. The abstract address can therefore be used for sharing objects. The concrete address(es) of an allocated memory object may change due to compaction. To this end, we point out another difference between the abstract and the concrete

address space. Over time, they may both get fragmented. The fragmentation of the concrete space presents a problem since upon an allocation request the memory management system must provide a sufficiently large contiguous memory range. In the case of the abstract address space, a single address is used per memory object, independently of its size. Hence, upon an allocation request, the memory management system needs to find only a single unused abstract address. We can achieve this within a constant-time bound, without keeping the abstract address space compact.

2.4 Size-Classes

For administration of the concrete address space, we adopt the approach set for Metronome [2, 3, 1].

The following ingredients describe the organization of the concrete address space.

- **Pages:** The memory is divided into units of a fixed size P , called pages. For example, in our implementation each page has a size $P = 16\text{KB}$.
- **Page-blocks:** Each used page is subdivided into page-blocks. All page-blocks in one page have the same size. In total, there are n predefined page-block sizes S_1, \dots, S_n where $S_i < S_j$ for $i < j$. Hence the maximal page-block size is S_n .
- **Size-classes:** Pages are grouped into size-classes. There are n size-classes (just as there are n page-block sizes). Let $1 \leq i \leq n$. Each page with page-blocks of size S_i belongs to the i -th size-class. Furthermore, each size-class is organized as a doubly-circularly-linked list.

Every allocation request is handled by a single page-block. When an allocation request `malloc(size)` arrives, CF determines the best fitting page-block size S_i and inserts the object into a page-block in a page that belongs to size-class i . The best fitting page-block size is the unique page-block size S_i that satisfies $S_{i-1} < \text{size} \leq S_i$.

If a used page becomes free upon deallocation, then the page is removed from its size-class and can be reused in any possible size-class.

Figure 3 shows an exemplary view of the organization of the concrete address space: There are three size-classes: in one of them there are two pages, in the other two there is a single page per class.

2.5 Fragmentation

The size-classes approach is exposed to several types of fragmentation: page-block-internal fragmentation,

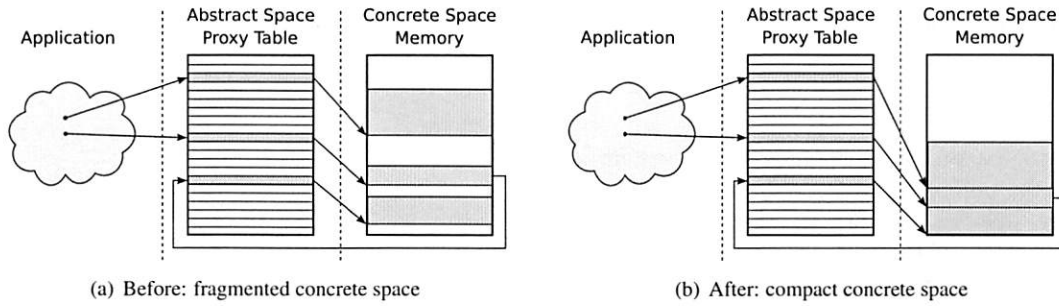


Figure 2: Compaction

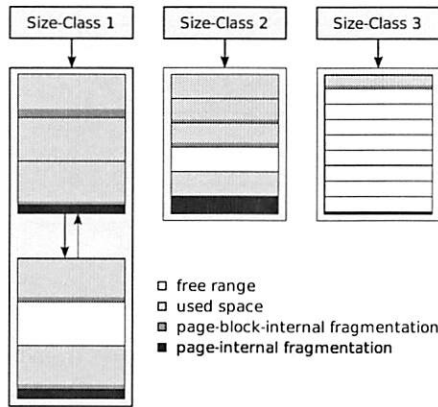


Figure 3: Size-classes

page-internal fragmentation, and size-external fragmentation [2]. We will briefly discuss each of them and the impact they have on our design decisions. Figure 3 shows the different types of fragmentation as well.

2.5.1 Page-Block-Internal Fragmentation

Page-block-internal fragmentation is the unused space at the end of a page-block. Given a page p in size class i (the page-blocks in p have size S_i), let b_j for $j = 1, \dots, B_p$ be the page-blocks appearing in the page p , where $B_p = P \text{ div } S_i$. For a page-block b_j we define $\text{used}(b_j) = 1$ if b_j is in use, and $\text{used}(b_j) = 0$ otherwise. We also write $\text{data}(b_j)$ for the amount of memory of b_j that is allocated. The page-block-internal fragmentation for the page p is calculated as

$$F_B(p) = \sum_{j=1}^{B_p} \text{used}(b_j) \cdot (S_i - \text{data}(b_j)).$$

One can also calculate the total page-block-internal fragmentation in the memory by summing up the page-block-internal fragmentation for each page.

The total page-block-internal fragmentation can be bounded by a factor f if the page-block sizes are chosen carefully. Namely, Berger et al. [4] suggest the following ratio between adjacent page-block sizes:

$$S_k = \lceil S_{k-1}(1 + f) \rceil \quad (1)$$

for $k = 2, \dots, n$. The size of the smallest page-blocks S_1 and the parameter f can be chosen program-specifically. Bacon et al. [2] propose a value for the parameter $f = 1/8$, which leads to minor size differences for smaller size-classes and major size differences for larger size-classes.

2.5.2 Page-Internal Fragmentation

Page-internal fragmentation is the unused space at the end of a page. If all possible page-block sizes S_1, \dots, S_n are divisors of the page size P , then there is no page-internal fragmentation in the system. However, if one uses Equation (1) for the choice of page-block sizes, then one also has to acknowledge the page-internal fragmentation. For a page p in size-class i , it is defined as

$$F_P(p) = P \bmod S_i.$$

The total page-internal fragmentation is the sum of $F_P(p)$ taken over all used pages p .

2.5.3 Size-External Fragmentation

Size-external fragmentation measures the unused space in a used page. This space is considered fragmented or “wasted” because it can only be used for allocation requests in the given size-class. For example, let p be a page in size-class i with $S_i = 32B$. If only one page-block of p is allocated, then there is $P - 32B$ unused memory in this page. If no more allocation requests arrive for this size-class, then this unused memory can never be used again. In such a situation an object of size

32B consumes the whole page. The size-external fragmentation of a page p in size-class i is bounded by

$$F_S(p) = P - S_i.$$

The total size-external fragmentation in the system is bounded by the sum of $F_S(p)$, over all pages.

The more size-classes there are in the system, the less page-block-internal fragmentation occurs, but therefore the size-external fragmentation may grow. Hence, there is a trade-off between page-block-internal and size-external fragmentation, which must be considered when defining the size-classes.

2.6 The Compaction Algorithm

The compaction algorithm of CF behaves as follows. Compaction is performed after deallocation in the size-class affected by the deallocation request. It implies movement of only one memory object in the affected size-class. Before presenting the algorithm, we state two invariants and two rules that are related to our compaction strategy. Recall that each size-class is a doubly-circularly-linked list.

INVARIANT 1. *In each size-class there exists at most one page which is not full.*

INVARIANT 2. *In each size-class, if there is a not-full page, then this is the last page in the size-class list.*

The compaction algorithm acts according to the following two rules.

RULE 1. *If a memory object of a full page p in a size-class gets freed, and there exists no not-full page, then p becomes the not-full page of its size-class and it is placed at the end of the size-class list.*

RULE 2. *If a memory object of a full page p in a size-class gets freed, and there exists a not-full page p_n in the size-class, then one memory object of p_n moves to p . If p_n becomes empty, then it is removed from the size-class.*

Not every deallocation request requires moving of a memory object. The cases when no moving is necessary are:

- The deallocated memory object is in the unique not-full page of the size-class. This case imposes no work except when the deallocated memory object is the only memory object in the page. Then the page is removed from the size-class.
- There is no not-full page in the size-class where deallocation happened. In this case only a fixed number of list-reference updates is needed in order that the affected page becomes the last page in the size-class list.

```
1 void compaction(size_class, affected_page) {
2   if (affected_page != last_page) {
3     if (is_full(last_page)) {
4       set_last(affected_page);
5     }
6   } else {
7     move(object, last_page, affected_page);
8     abstract_address_space_update(object);
9   }
10 }
11 }
```

Listing 1: The compaction algorithm

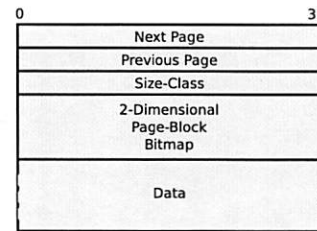


Figure 4: Page layout

Note that when a memory object moves from one page to another, then we need to perform a reference update in the abstract address space, in order that the abstract address of a memory object points to its new location.

The compaction algorithm is presented in Listing 1.

3 Details and Complexity

In this section we elaborate on the implementation details of CF. We start by discussing the details of the concrete address space management, i.e., administration of the page-blocks, pages, and size-classes. Next we describe the implementation and the complexity results for both the *moving* and the *non-moving* version of CF. At the end of this section we also explain the concept of partial compaction.

3.1 Managing Size-Classes

As mentioned above, we use pages of size 16KB. If needed, the page size can be modified for specific applications. The minimal page-block size S_1 in our benchmarks is 32B, but it can also be reduced to a smaller size. Successive page-block sizes are determined by Equation (1) taking $f = 1/8$.

In addition to the 16KB storage space, each page has a header with additional information used for the memory management. The layout of a page with all administrative information is illustrated in Figure 4.

The fields Next Page and Previous Page contain references to the next and the previous page within the size-class, respectively. These two references build the size-class list. The field Size-Class refers to the size-class instance of the page, which further refers to the head of the size-class list. Hence, we can directly access the head and therefore also the tail of the size-class list from any page, which is important for compaction.

A page consists of page-blocks, some of which are in use and some are free. For memory object allocation, we must find a free page-block in a page in constant time and for compaction, we must find a used page-block in a page in constant time. Therefore, the state of the page (the status of all page-blocks) is recorded in a two-dimensional bitmap, as shown in Figure 4. A bitmap of size 16×32 is enough to record the status of the page-blocks since we have at most 512 page-blocks per page. In addition, we need 16 more bits to record if at least one bit is set for each row of the bitmap. This additional bitstring is essential for constant-time access to a used page-block within the page and it is used to determine whether a page is full or not in constant time (one comparison). There are CPU instructions that find a set bit in a bitstring in constant time. These instructions are limited to bitstrings of length 32 (on a 32-bit CPU), which is the reason why we use such a two-dimensional bitmap. In order to get a used (respectively free) page-block we first find a set bit in the additional bitstring, and then get a set bit in the corresponding row of the bitmap.

The free pages are organized in a special LIFO list. Since all pages are arranged contiguously in memory, no list initialization is necessary. If the LIFO list is empty and a free element is required, the next unused element of the contiguous space is returned (if such an element exists). We refer to this list construction as *free-list*.

Note that the administrative information of a page, as shown in Figure 4, takes 78B out of the 16KB page memory. Hence, the memory overhead is less than 0.47%.

3.2 Moving Implementation

In this version, memory objects are moved in physical memory during compaction.

The abstract address space is implemented in a contiguous piece of memory. The free entries of the abstract address space are organized in a free-list.

The concrete address space is organized as described in Section 3.1. Each page is implemented as a contiguous piece of memory as well. Moreover, each page-block contains an explicit reference to its abstract address in the abstract address space. This reference is located at the end of the page-block. In the worst case, it occupies 12.5% of the page-block size. These backward references allow us to find in constant time the abstract ad-

```
1 void **cfm_malloc(size) {
2   page = get_page_of_size_class(size);
3   page_block = get_free_page_block(page);
4   return create_abstract_address(page_block);
5 }
```

Listing 2: Allocation - moving version

dress of a memory object of which we only know its concrete address. Therefore they are essential for constant-time update of the abstract address space during compaction.

We next present the allocation, deallocation, and dereferencing algorithms and discuss their complexity. The algorithm for allocation `cfm_malloc` is presented in Listing 2. The function `get_page_of_size_class` returns a page of the corresponding size-class in constant time: if all pages in the size-class are full, then with help of the free-list of free pages, we get a new page; otherwise the not-full page of the size-class is returned. Hence this function needs constant time. The function `get_free_page_block` takes constant time, using the inverse of the two-dimensional bitmap of a page. Declaring a page-block used is just a bit-set operation. As mentioned above, the free abstract addresses are organized in a free-list, so the function `create_abstract_address` takes constant time. As a result, `cfm_malloc` takes constant time, i.e., $\Theta(\text{cfm_malloc}(\text{size})) = \Theta(1)$.

The deallocation algorithm `cfm_free` is shown in Listing 3. The function `get_page_block` takes constant time, since it only accesses the memory location to which the abstract address refers. The function `get_page` takes several arithmetic operations, i.e., constant time. Namely, pages are linearly aligned in memory so for a given page-block we can calculate the starting address of its page. The function `get_size_class` is executed in constant time, since every page contains a field Size-Class. The function `set_free_page_block` changes the value of a single bit in the bitmap, so it also requires constant time and `add_free_abstract_address` amounts to adding a new element to the corresponding free-list, which is done in constant time too. Removing a page from a size-class `remove_page` requires also constant time: first a page is removed from the size-class list; then it is added to the free-list of empty pages. Therefore, the complexity of `cfm_free` equals the complexity of the compaction algorithm.

The complexity of the compaction algorithm, `compaction`, is linear in the size of the page-blocks in the corresponding size-class since it involves moving a memory object. Note that the complexity of the abstract address space update is constant, due to the direct

```

1 void cfm_free(abs_address) {
2   page_block = get_page_block(abs_address);
3   page = get_page(page_block);
4   size_class = get_size_class(page);
5   set_free_page_block(page, page_block);
6   add_free_abstract_address(abs_address);
7   if (page == empty) {
8     remove_page(size_class, page);
9   }
10  else {
11    compaction(size_class, page);
12  }
13 }

```

Listing 3: Deallocation - moving version

reference from page-blocks to abstract addresses.

Hence, the worst-case complexity of `cfm_free` is linear in the size of the page-block: $\mathcal{O}(\text{cfm_free}(\text{abs_address})) = \mathcal{O}(s)$ for s being the size of page-blocks in the size-class where `abs_address` refers to. Thus, for a fixed size-class, we have constant complexity.

In this moving implementation, the physical location of a memory object is accessed by dereferencing an abstract pointer. The dereferencing contains a single line of code `*(abs_address + offset);` given an abstract pointer `(abs_address, offset)`.

To conclude, the only source of non-constant (linear) complexity in the moving implementation is the moving of objects during compaction. In an attempt to lower this bound by a constant factor, we implement the non-moving version.

3.3 Non-Moving Implementation

We call this implementation non-moving, since memory objects do not change their location in physical memory throughout their lifetime, even if compaction is performed.

In the non-moving implementation, the abstract address space is still a contiguous piece of memory. However, we no longer use the free-list for administrating free abstract addresses, since now there is an implicit mapping from the memory objects to the abstract addresses. We will elaborate on this in the next paragraph. The implicit references (from memory objects to abstract addresses) are used for constant-time updates of the abstract address space.

First, let us explain the difference in the implementation of the concrete address space. The concrete address space is managed by a virtual memory. The virtual memory consists of blocks of equal size. The physical memory is contiguous and correspondingly divided into block-frames of equal size. For further reference we

denote this size by s_b . These blocks and block-frames must not be confused with the page-blocks: they are global, for the whole memory, above the page concept. Therefore, each block-frame is directly accessible by its ordinal number. The free block-frames are also organized in a free-list. The abstract address space is pre-allocated, and contains as many potential abstract addresses as there are block-frames. A unique abstract address corresponds to a memory object m : the abstract address at position k in the abstract address space, where k is the ordinal number of the first block-frame of m in the physical memory. This way, we do not need an explicit reference stored in each page-block, thus we avoid the space overhead characteristic to the moving implementation. Moreover, getting a free abstract address is immediate, as long as we can allocate a block-frame in physical memory.

A block table records the mapping from virtual memory blocks to memory block-frames. In our implementation, the block table is distributed over the pages, which simplifies the page-specific operations. The organization of the memory in this implementation is shown in Figure 5.

Objects are allocated in contiguous virtual memory ranges, but actually stored in arbitrarily distributed physical memory block-frames. We still use the concepts of pages, size-classes and page-blocks, with some differences. A page is now a virtual page: It does not contain any data; in its Data segment it contains the part of the block table that points to the actual data in the physical memory. Moreover, for administration purposes all page-block sizes are multiples of the unique block size. Hence, each page-block consists of an integer number of blocks. This implies that we can not directly use Equation (1), we need to round-up the page-block sizes to the next multiple of the block size.

The allocation algorithm is shown in Listing 4. In comparison to the moving implementation, we have now a loop that handles the allocation block-frame-wise. Note that `number_of_blocks(page_block)` is constant for a given size-class. Getting a free block-frame, and creating a corresponding block-table entry, takes constant time. Therefore, the complexity of the allocation algorithm is linear in the number of block-frames in a page-block, i.e., $\Theta(\text{cfm_malloc}(\text{size})) = \Theta(n)$ where $n = s/s_b$ for s the page-block size of the size-class. Again, this means constant complexity in the size-class.

The function `create_abstract_address` in the non-moving implementation uses the implicit references from memory objects to abstract addresses.

Listing 5 shows the deallocation algorithm.

In comparison to the moving implementation, we have to free each block-frame in the memory occupied by

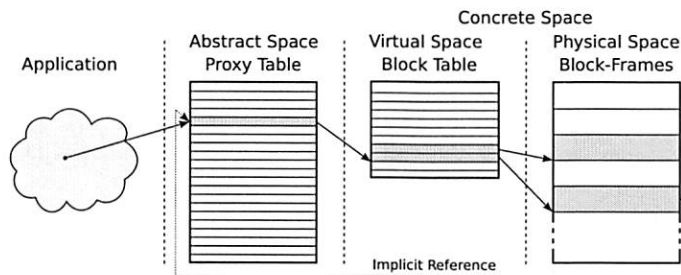


Figure 5: Memory layout of the non-moving implementation

```

1 void **cfnm_malloc(size) {
2   page = get_page_of_size_class(size);
3   page_block = get_free_page_block(page);
4   for (i = 1 to number_of_blocks(page_block)) {
5     block_frame = get_free_block_frame();
6     add_to_block_table(page,
7       page_block, block_frame);
8   }
9   return create_abstract_address(page_block);
10 }

```

Listing 4: Allocation - non-moving version

the freed memory object. This requires a loop with as many iterations as there are block-frames for the memory object. As above, there are $n = s/s_b$ block-frames, where s is the size of the size-class and s_b the size of the block-frame. Moreover, the compaction algorithm is implemented differently: memory objects are only virtually moved, by updating the block table. This is still linear in the size of the size-class, but we achieve a constant speed-up: it actually takes n updates of the block table. As a result, the complexity of the deallocation algorithm in the non-moving implementation is $\Theta(\text{cfnm_free}) = \Theta(n)$, which is again constant for a given size-class.

Finally, we consider the dereference algorithm. It provides direct access to a memory location corresponding to a given abstract pointer (abs_address , offset). Dereferencing takes constant time, with several more calculations than in the moving implementation. The algorithm is shown in Listing 6. In the code, s_b stands for the size of a block, s_b .

We conclude that the non-moving implementation achieves a constant speed-up of the compaction algorithm, while the complexity of the allocation algorithm grows from constant to linear. The complexity of deallocation and dereference is the same in both implementations.

```

1 void cfnm_free(abs_address) {
2   page_block = get_page_block(abs_address);
3   page = get_page(page_block);
4   size_class = get_size_class(page);
5   for (i = 1 to number_of_blocks(page_block)) {
6     block_frame =
7       get_block_frame(page_block, i);
8     add_free_block_frame(block_frame);
9   }
10  set_free_page_block(page, page_block);
11  add_free_abstract_address(abs_address);
12  if (page == empty) {
13    remove_page(size_class, page);
14  }
15  else {
16    compaction(size_class, page);
17  }
18 }

```

Listing 5: Deallocation - non-moving version

```

1 void *cfnm_dereference(abs_address, offset) {
2   return (*(abs_address + (offset div s_b))
3     + (offset mod s_b));
4 }

```

Listing 6: Dereference - non-moving version

3.4 Partial Compaction

Up to now we always considered the very strong aim of “always compact size-classes”, i.e., our invariant was that at any moment in time in each size-class at most one page is not full. We now relax this invariant by allowing for a given number of not-full pages per size-class. Varying the number of allowed not-full pages per size-class, max_nr_nf_pages , results in various levels of compaction. For example, $\text{max_nr_nf_pages} = 1$ means always compact memory (as described above), but therefore larger compaction overhead, whereas high values of max_nr_nf_pages lead to faster memory management, for the price of higher fragmentation. This way we formalize, control, and implement the trade-off between temporal performance and memory fragmentation.

We implement this new approach as follows. A size-class instance now consists of three fields: `head_full_pages`, `head_not_full_pages`, and `nr_not_full_pages`. Therefore it consists of two doubly-circularly-linked lists: one containing the full pages and another one containing the not-full pages. The initial value of `nr_not_full_pages` is zero and it never exceeds a pre-given number, `max_nr_nf_pages`. If all pages are full, then allocation produces a new not-full page, so `nr_not_full_pages` is incremented. In case a not-full page gets full after allocation, it is moved to the list of full pages, and `nr_not_full_pages` is decremented. If deallocation happens on a page-block that is in a not-full page, no compaction is done. If it happens on a page-block which is in a full page, then compaction is called if `nr_not_full_pages` = `max_nr_nf_pages`. Otherwise, no compaction is done, the affected page is moved from the list of full pages to the list of not-full pages, and `nr_not_full_pages` is incremented.

For better average-case temporal and spatial performance, we keep pages that are more than half-full at the end of the not-full list, and pages that are at most half-full at the head of the list. Allocation is served by the last page of the not-full list, which may increase the chances that this page gets full. Used page-blocks that need to be moved because of compaction are taken from the first page of the not-full list, which might increase the chances that this page gets empty. Note that, for the best possible spatial performance, it would be better to keep the not-full list sorted according to the number of free page-blocks in each non-full page. However, inserting a page in a sorted list is too time-consuming for our purposes and can not be done in constant time.

It should be clear that each deallocation when `nr_not_full_pages` < `max_nr_nf_pages` takes constant time, i.e., involves no compaction. However, this guarantee is not very useful in practice: given a mutator we can not find out (at compile time) the maximal number of not-full pages it produces. Therefore we describe another guarantee.

Given a size-class, let n_f count deallocations for which no subsequent allocation was done. Initially, $n_f = 0$. Whenever deallocation happens, n_f is incremented. Whenever allocation happens, n_f is decremented, unless it is already zero. We can now state the following guarantee.

PROPOSITION 1. *Each deallocation that happens when $n_f < \text{max_nr_nf_pages} - 1$ takes constant time in the CF moving implementation, i.e., it involves no compaction.*

Namely, a simple analysis shows that $\text{nr_not_full_pages} \leq n_f + 1$ is an invari-

```
1 int *value;
2 value = malloc(40);
3 value++;
4 print(*value);
```

Listing 7: Standard C pointers example

```
1 struct abs_pointer value;
2 value.abs_address = malloc(40);
3 value.offset = 0;
4 value.offset += 4;
5 print(dereference(value.abs_address,
6   value.offset));
```

Listing 8: Abstract pointers example

ant for a given size-class. It holds initially since then `nr_not_full_pages` = n_f = 0. Each allocation and deallocation keeps the property valid. Therefore, if a deallocation happens when $n_f < \text{max_nr_nf_pages} - 1$, then `nr_not_full_pages` < `max_nr_nf_pages` and hence compaction is not called.

Program analysis can be used to determine the maximum value of n_f for a given mutator at compile time. More advanced program analysis (e.g. analysis of sequences of allocations and deallocations) might provide us with a stronger guarantee than the one above. Employing program analysis is one of our future-work aims. The effect of partial compaction can be seen in the experiments in Section 5.

3.5 Pointer Arithmetic

Since we make the distinction between abstract and concrete address space, our `malloc` function returns a reference to an abstract address, instead of a reference to a memory location. Therefore, pointer arithmetic needs adjustment, so that it fits our model. In order to enable standard pointer arithmetic, we need the structure of an abstract pointer. It contains a field `abs_address` and a field `offset`. Consider the example of C code in Listing 7. The same is achieved in CF by the code presented in Listing 8.

A virtual machine (e.g. a Java VM) can encapsulate the abstract pointer concept. C programs which use CF instead of a conventional memory management system have to be translated into code that uses the abstract pointer concept. An automatic translation tool is left for future work.

On an Intel architecture running Linux, GCC translates a standard pointer dereference (including offset addition) to 6 assembler instructions, whereas a CFM abstract pointer dereference results in 7 assembler instruc-

tions (one additional pointer dereference instruction is needed). A CFNM abstract pointer dereference results in 11 assembler instructions. The additional 5 assembler instructions consist of one dereference operation and 4 instructions that calculate the memory target address and move the pointer to that address.

4 Related Work

In this section we briefly discuss existing memory management systems (mostly for real time). We distinguish explicit and implicit memory management systems and consider both non-compacting real-time, and non-real-time memory management systems. Jones [6] maintains an extensive online bibliography of memory management publications.

4.1 Explicit Memory Management

There are several established explicit memory management systems: First-fit [7], Best-fit [7], Doug Lea's allocator (DL) [8], Half-fit [11], and Two-level-segregated-fit (TLSF) [10]. A detailed overview of the explicit memory management systems can be found in [9, 13].

First-fit and Best-fit are sequential fit allocators, not suitable for real-time applications. In the worst case, they scan almost the whole memory in order to satisfy an allocation request. DL is a non-real-time allocator used in many systems, e.g. in some versions of Linux.

Half-fit and TLSF offer constant response-time bounds for allocation and deallocation. However, both approaches may suffer from unbounded memory fragmentation.

None of the above mentioned algorithms perform compaction. Instead, they attempt to fight fragmentation by clever allocation, and therefore can not give explicit fragmentation guarantees.

In Section 5 we present a comparison of the CF implementations with First-fit, Best-fit, DL, Half-fit, and TLSF.

4.2 Implicit Memory Management

We elaborate on two established implicit real-time memory management systems: Jamaica [14] and Metronome [2].

Jamaica splits memory objects into fixed-size blocks that can be arbitrarily located in memory and connected in a linked list (or tree) structure. Allocation and deallocation achieve the same bounds like our non-moving implementation. Dereferencing in Jamaica involves going through the list of memory object blocks, therefore it takes linear (or logarithmic) time in the size of the object.

Compaction is not needed for Jamaica, since memory objects do not occupy contiguous pieces of memory.

Metronome is a time-triggered garbage collector. As mentioned above, we adapt some concepts like pages and size-classes from Metronome. Compaction in Metronome is part of the garbage collection cycles. The time used for compaction is estimated to at most 6% of the collection time [2].

5 Experiments and Results

In this section, we benchmark the moving (CFM) and non-moving (CFNM) implementations as well as the partial compaction strategy of CF in a number of experiments. Moreover, we compare both CF implementations with the dynamic memory management algorithms First-fit, Best-fit, DL, Half-fit, and TLSF. The implementations of First-fit, Best-fit, Half-fit, and TLSF we borrow from Masmano et al. [9]. We took the original implementation of DL from Doug Lea's web page [8].

5.1 Testing Environment

We have performed processor-instruction measurements of our algorithms on a standard Linux system, and bare-metal execution-time measurements on a Gumstix connex400 board [5] running a minimal hardware abstraction layer (HAL).

Processor-instruction measurements eliminate interferences like cache effects. For measurement purposes, our mutators are instrumented using the ptrace [12] system call. The processor-instruction and execution-time measurements are almost the same except that the former are cleaner, free of side effects. For this reason, we present the processor-instruction results only.

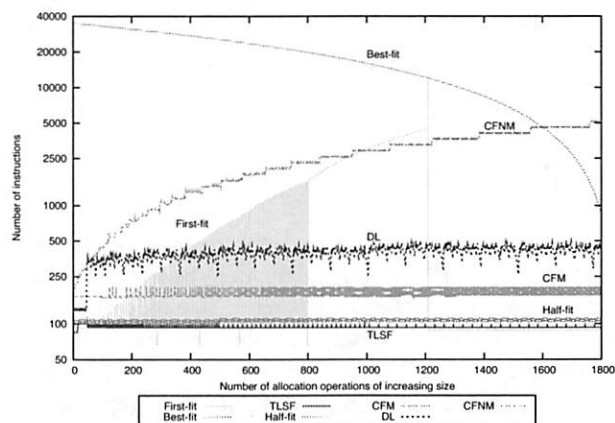
Our mutators provide synthetic workloads designed to create worst-case and average-case scenarios. We have not obtained standardized macrobenchmark results for lack of an automatic code translator or virtual machine implementation that incorporate the abstract pointer concept.

5.2 Results: Incremental Tests

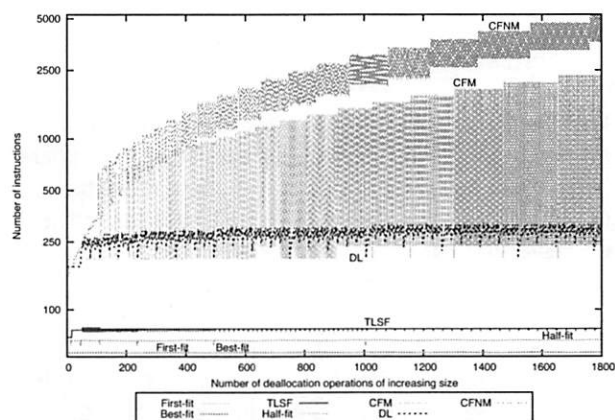
In this benchmark, we run a mutator with incremental behavior: it allocates memory objects of increasing size starting from 8B increasing by 4B until the memory gets full at 7MB. Then, it deallocates each second object. We measure this process in the deallocation experiments. Finally, the mutator allocates the deallocated objects once more. We measure this process in the allocation experiments. In Figure 6, the x -axes show the number of invoked memory operations, whereas the y -axes represent

the corresponding measured number of executed instructions.

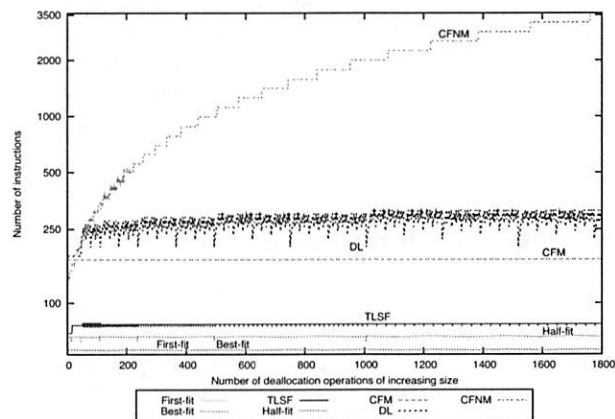
Figure 6(a) shows the number of processor instruc-



(a) Allocation



(b) Deallocation with compaction



(c) Deallocation, partial compaction

Figure 6: Incremental microbenchmark

tions for allocation. The behavior of First-fit and Best-fit is highly unpredictable. DL appears more bounded. Half-fit and TLSF perform allocation operations fast and in constant time. The behavior of the CF implementations is according to our theoretical results: constant for CFM and linear for CFNM. Note that the y -axes of the graphs have logarithmic scale. Both CF implementations are bounded but slower than Half-fit and TLSF due to the additional administrative work for (potential) compaction. CFM is as fast as DL, and faster than First-fit and Best-fit. The average number of instructions for allocation with CFM is 169.61, the standard deviation is 8.63.

The deallocation benchmark, with full compaction, is presented in Figure 6(b). All algorithms except CF perform deallocation in constant time by adding the deallocated memory range to a data structure that keeps track of the free memory slots. CF performs compaction upon deallocation, and therefore takes linear time (in the size of the memory object) for deallocation. The overhead of performing compaction leads to longer execution time, but both CF implementations are bounded and create predictable memory. For the given block-frame size of 32B, CFNM does not perform better than CFM since returning blocks to the free-list of free block-frames takes approximately the same time as moving a whole memory object. Experiments showed that the minimum block-frame size for which deallocation in CFNM is faster than in CFM is 80B.

Using the partial compaction strategy results in constant deallocation times for CFM, as shown in Figure 6(c). Note that this graph shows the same picture as Figure 6(b) except for CFM and CFNM where partial compaction is applied. The compaction bounds for partial compaction are set sufficiently wide to avoid compaction. CFNM shows a step function with tight bounds per size-class. The average number of instructions for deallocation with CFM and partial compaction is 185.91, the standard deviation is 16.58.

5.3 Results: Rate-Monotonic Tests

In the rate-monotonic scheduling benchmarks, we use a set of five periodic tasks resembling a typical scenario found in many real-time applications. Each task allocates memory objects of a given size, and deallocates them when the task's holding time expires. Three of the tasks allocate larger objects and have long holding times, the other two allocate small objects and have short holding times. The tasks have various periods and deadlines. They are scheduled using a rate-monotonic scheduling policy. Since the different tasks create a highly fragmented memory, this benchmark represents a memory fragmentation stress test.

For better readability, the y -axes of the graphs show the cumulative number of instructions, i.e., the sum of the number of executed instructions for all operations starting from the first invoked operation up to the currently invoked one. The x -axes show the number of invoked operations, as before. Note that a linear function represents memory operations that take constant time.

The allocation measurements are presented in Figure 7(a). Best-fit is highly unpredictable. Half-fit and TLSF are constant and fast. CFM is also constant, faster than DL, but slightly slower than Half-fit and TLSF. On average, a CFM allocation request takes 169.61 instructions with a standard deviation of 8.63.

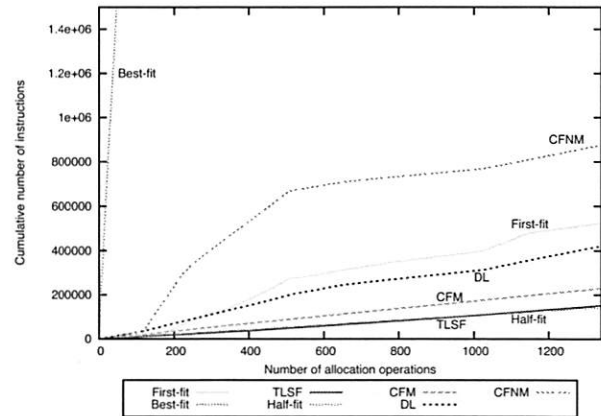
Figure 7(b) shows the deallocation measurements. The differences in growth of the CFM curve correspond to compaction. During the first 550 deallocation operations CFM has to perform a lot of compaction operations but afterwards no compaction is necessary. The total runtime is shorter than the time needed for DL. CFNM takes linear time in the size of the memory object even if there is no compaction performed. The curve reflects this property.

Applying partial compaction leads to constant-time deallocation with CFM and makes it fast and more predictable, as shown in Figure 7(c). This graph shows the same picture as Figure 7(b) except for CFM and CFNM where partial compaction is applied. In order to apply partial compaction we have used the following compaction bounds on the 46 size-classes in the system: In the size-classes 15-18 and 28-29, two not-full pages are allowed. In the size-classes 19-27, we allow for three not-full pages. All other size-classes can have at most one not-full page. The mean number of instructions for CFM deallocation with partial compaction is 171.61, the standard deviation is 5.09.

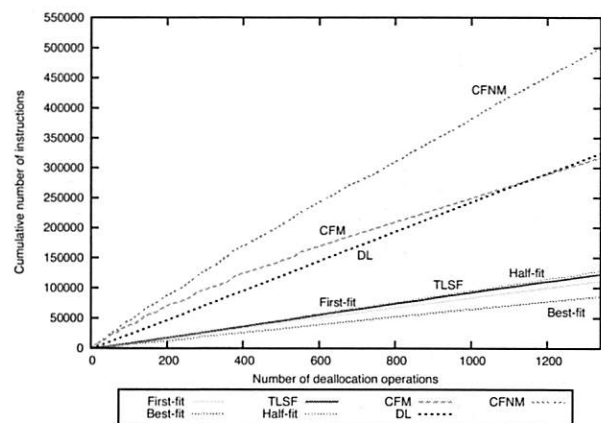
5.4 Results: Fragmentation Tests

Our final experiments measure fragmentation. We compare CFM (with partial compaction) with TLSF, since the latter is considered the best existing real-time memory management system in terms of fragmentation [9]. The results are shown in Figure 8. The numbers next to CFM, e.g. CFM 3, denote the maximal number of not-full pages allowed in each size-class.

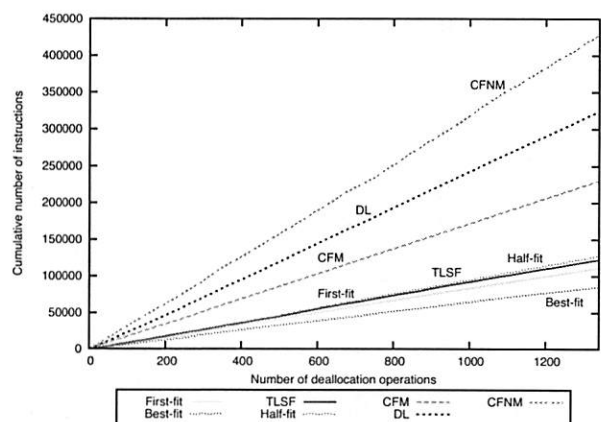
For the experiments we have used a mutator that allocates the whole memory using memory objects of size 20B-100B. Before we run the fragmentation test around 20% of the number of allocated objects is freed. The memory holes are randomly distributed throughout the memory. The fragmentation tests count how many objects (y -axis) of size 20B-16000B (x -axis) are still allocatable by each memory management system. CFM obviously deals with fragmentation better than TLSF, even



(a) Allocation



(b) Deallocation compaction



(c) Deallocation, partial compaction

Figure 7: Rate-monotonic microbenchmark

if we allow up to nine not-full pages in each size-class. Moreover, the fragmentation in CFM is fully controlled and predictable.

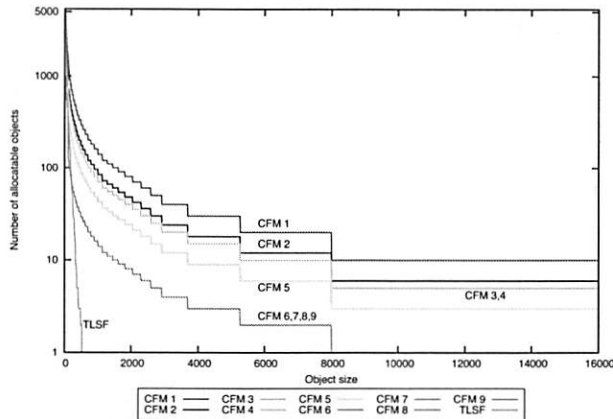


Figure 8: Fragmentation

6 Discussion and Conclusion

Compact-fit is an explicit real-time memory management system that handles fragmentation through compaction like some implicit real-time memory management systems do. Its main new contribution is predictable response times in combination with predictable memory fragmentation.

We have designed and implemented two versions of Compact-fit. Allocating an object takes constant time in the moving implementation and linear time (in the object's size) in the non-moving implementation. Deallocating an object takes linear time (in its size) in both implementations. If no compaction occurs, deallocating takes constant time in the moving implementation. Dereferencing takes constant time in both implementations.

Hence, we provide tight bounds on the response times of memory operations. Moreover, we keep each size-class (partially) compact, i.e., we have predictable memory. Hence, unlike the other existing real-time memory management systems that do not fully control fragmentation, our compacting real-time memory management system is truly suitable for real-time and even safety-critical applications.

Finally, another real-time characteristic of our memory management system is the constant initialization time. This is achieved using the free-list concept for all resources (abstract addresses, pages, block-frames, etc.) that need initialization.

The experiments validate our asymptotic complexity results. Due to more administrative work related to compaction, our system is slightly slower than the existing systems with real-time response bounds.

There are several possible improvements to our design and implementation that we leave for future work. In our present work, the abstract address space is stati-

cally pre-allocated to fit the worst case. For less memory overhead, we could implement a dynamic abstract address space allocation by using the pages from the concrete address space also for storing abstract addresses. Moreover, the present implementation allows for memory objects of size at most 16KB, the size of a page. Arraylets [3] can be used in order to handle objects of larger size. Other topics for future work are concurrency support, program analysis for determining optimal partial compaction bounds and needed amount of abstract addresses, and allocatability analysis.

Acknowledgments

This work is supported by a 2007 IBM Faculty Award, the EU ArtistDesign Network of Excellence on Embedded Systems Design, and the Austrian Science Fund No. P18913-N15.

References

- [1] BACON, D. F. Realtime garbage collection. *Queue* 5, 1 (2007), 40–49.
- [2] BACON, D. F., CHENG, P., AND RAJAN, V. T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proc. LCTES (2003)*, ACM Press, pp. 81–92.
- [3] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proc. POPL (2003)*, ACM Press, pp. 285–298.
- [4] BERGER, E. D., MCKINLEY, K. S., BLUMOF, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. ASPLOS (2000)*, ACM Press, pp. 117–128.
- [5] GUMSTIX. Gumstix inc. <http://www.gumstix.org>.
- [6] JONES, R. The garbage collection page. <http://www.cs.ukc.ac.uk/people/staff/rej/gc.html>. The definitive on-line resource for garbage collection material.
- [7] KNUTH, D. E. *Fundamental Algorithms*, second ed., vol. 1 of *The Art of Computer Programming*. Addison-Wesley, 1973.
- [8] LEA, D. A memory allocator. *Unix/Mail*, 6/96, 1996.
- [9] MASMANO, M., RIPOLL, I., AND CRESPO, A. A comparison of memory allocators for real-time applications. In *Proc. JRES (2006)*, ACM press, Vol. 177, pp. 68–76.
- [10] MASMANO, M., RIPOLL, I., CRESPO, A., AND REAL, J. TLSF: A new dynamic memory allocator for real-time systems. In *Proc. ECRTS (2004)*, IEEE Computer Society, pp. 79–86.
- [11] OGASAWARA, T. An algorithm with constant execution time for dynamic storage allocation. In *Proc. RTCSA (1995)*, IEEE Computer Society, pp. 21–27.
- [12] PADALA, P. Playing with ptrace, part I. *Linux Journal* 2002, 103 (2002), 5.
- [13] PUAUT, I. Real-time performance of dynamic memory allocation algorithms. In *Proc. ECRTS (2002)*, IEEE Computer Society, pp. 41–49.
- [14] SIEBERT, F. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proc. CASES (2000)*, ACM Press, pp. 9–17.

Prefetching with Adaptive Cache Culling for Striped Disk Arrays

Sung Hoon Baek and Kyu Ho Park

Korea Advanced Institute of Science and Technology

shbaek@core.kaist.ac.kr, kpark@ee.kaist.ac.kr

Abstract

Conventional prefetching schemes regard prediction accuracy as important because useless data prefetched by a faulty prediction may pollute the cache. If prefetching requires considerably low read cost but the prediction is not accurate, it may or may not be beneficial depending on the situation. However, the problem of low prediction accuracy can be dramatically reduced if we efficiently manage prefetched data by considering the total hit rate for both prefetched data and cached data. To achieve this goal, we propose an adaptive strip prefetching (ASP) scheme, which provides low prefetching cost and evicts prefetched data at the proper time by using differential feedback that maximizes the hit rate of both prefetched data and cached data in a given cache management scheme. Additionally, ASP controls prefetching by using an online disk simulation that investigates whether prefetching is beneficial for the current workloads and stops prefetching if it is not. Finally, ASP provides methods that resolve both independency loss and parallelism loss that may arise in striped disk arrays. We implemented a kernel module in Linux version 2.6.18 as a RAID-5 driver with our scheme, which significantly outperforms the sequential prefetching of Linux from several times to an order of magnitude in a variety of realistic workloads.

1 Introduction

Prefetching is necessary to reduce or hide the latency between a processor and a main memory as well as between a main memory and a storage subsystem that consists of disks. Some prefetching schemes for processors can be applied to prefetching for disks by means of a slight modification. And many prefetching techniques that are dedicated to disks have been studied. We focus on disk prefetching, especially for striped disk arrays.

The frequently-addressed goal of disk prefetching is to make data available in a cache before the data is consumed; in this way, computational operations are overlapped with the transfer of data from the disk. The other goal is to enhance the disk throughput by aggregating multiple contiguous blocks as a single request. Prefetching schemes for a single disk cause problems is used in striped disk arrays. There is a need for a special scheme

for multiple disks, in which the characteristics of striped disk arrays [7] are considered.

1.1 Five Problems

The performance disparity between processor speed and the disk transfer rate can be compensated for via the disk parallelism of disk arrays. Chen et al. [7] described six types of disk arrays and termed them redundant disk arrays of independent disks (RAID). In these arrays, blocks are striped across the disks, and the striped blocks provide the parallelism of multiple disks, thereby improving access bandwidth. Many RAID technologies have focused on the following: reliability of RAID [1, 5, 16, 34, 40], the write performance [3, 12, 46], multimedia streaming with a disk array [15, 21, 27], RAID management [45, 47], and so on [20, 41].

However, prefetching schemes for disk arrays have been rarely studied. Some offline prefetching schemes (described in Section 1.2.3) take load balancing among disks into account, though they are not practical since they require complete knowledge of future I/O references.

For multiple concurrent reads, the striping scheme of RAID resolves the load balancing among disks [7]. The greater number of concurrent reads implies more evenly distributed reads across striped blocks. As a result, researchers are forced to address five new problems that substantially affect the prefetching performance of striped disk arrays as follows:

1.1.1 Parallelism

If the prefetch size is much less than the stripe size and the number of concurrent reads is much less than the number of the member disks that compose a striped disk array, some of disks become idle, thereby losing parallelism of the disks. This case exemplifies what we call *parallelism loss*.

In sequential prefetching (described in Section 1.2.4), a large prefetch size laid across multiple disks can prevent parallelism loss. For the worst case of a single stream, the prefetch size must be equal to or larger than the stripe size. However, this method gives rise to prefetching wastage [10] for multiple concurrent sequential reads, which are common in streaming servers. A streaming server that serves tens of thousands of clients,

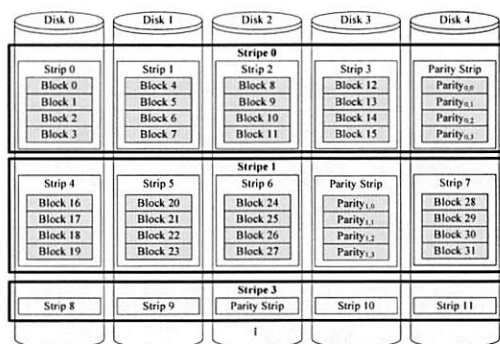


Figure 1: The data organization and terminologies of a RAID-5 array

requires several gigabytes of prefetching memory. If the server does not have an over-provisioned memory, thrashing occurs as the prefetched pages are replaced too early by massive prefetches before they are used.

1.1.2 Independency

Traditional prefetching schemes have focused on parallelism or the load balance of disks. However, this paper reveals that the *independency* of disks is more important than *parallelism* for concurrent reads of large numbers of processes in striped disk arrays, whereas parallelism is only significant when the number of concurrent accesses is roughly less than the number of member disks.

A stripe is defined by the RAID Advisory Board [39] as shown in Fig. 1, which illustrates a RAID-5 array consisting of five disks. The stripe is divided by the strips. Each strip is comprised of a set of blocks.

Figure 2(a) shows an example of *independency loss*. The prefetching requests of conventional prefetching schemes are not aligned in the strip; therefore, a single prefetching request may be split across several disks. In Fig. 2(a), three processes request sequential blocks for their own files. A preset amount of sequential blocks are aggregated as a single prefetching request by the prefetcher. If each prefetch request is not dedicated to a single strip, it is split across two disks, and each disk requires two accesses. For example, if a single prefetch request is for Block 2 to Block 5, the single prefetch request generates two disk commands that correspond to Block 2 & 3 belonging to Disk 0 and Block 4 & 5 belonging to Disk 1. This problem is called *independency loss*. In contrast, if each prefetching request is dedicated to only one disk, as shown in Fig. 2(b), independency loss is resolved.

To resolve independency loss, each prefetching request must be dedicated to a single strip and not split to multiple disks. The strip size is much less than the stripe size and, as a result, suffers parallelism loss for small numbers of concurrent reads. The two problems,

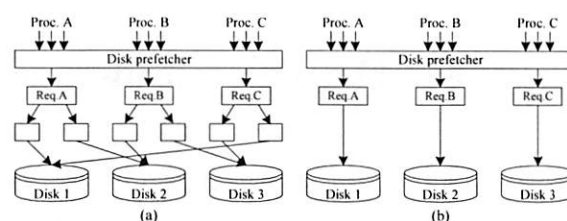


Figure 2: (a) Independency loss: prefetch requests are split across multiple disks so that each prefetch requires two disk accesses. (b) No independency loss: each prefetch request is dedicated to one disk.

independency loss and parallelism loss, conflict with one other. In the traditional prefetching schemes, if one problem is resolved, the other arises.

We propose an adaptive strip prefetching (ASP) scheme that eliminates almost all demerits of a new strip prefetching scheme, as discussed in Section 2.1. The proposed strip prefetching prefetches all blocks of a strip of a striped disk array on a cache miss. The strip is a set of contiguous blocks dedicated to only one disk, ASP including strip prefetching resolves independency loss. Parallelism loss is resolved by combining ASP with our earlier work, massive stripe prefetching (MSP) [2], which maximizes parallelism for sequential reads of a small number of processes.

1.1.3 Non-Sequential Read and Small Files

Sequential prefetching is a widely used practical prefetching scheme. However, it is never beneficial to non-sequential reads, although they may exhibit some spatial locality. In addition, the prefetch size of sequential prefetching is restricted by file size. Although a single process sequentially reads contiguous small files, the small prefetch size of sequential prefetching causes parallelism loss. Such workloads are common in real systems but there is no scheme that has the advantages of being; (1) beneficial to them, (2) practical for real systems, (3) of low overhead, (4) transparent to applications, and (5) convenient to use.

ASP improves the performance of such workloads because strip prefetching exploits the principle of the spatial locality by implicitly prefetching data that is likely to be referenced in the near future.

1.1.4 Prefetched Data Management

Traditional prefetching schemes regard prediction accuracy as important because there is no efficient and practical cache management scheme for uselessly prefetched data that pollute the cache. Strip prefetching requires considerably low read cost but its prefetching prediction is not so accurate. Consequently, the overall throughput of strip prefetching may or may not be beneficial de-

pending on the situation. However, the problem of low prediction accuracy can be dramatically reduced if we efficiently manage prefetched data by considering the total hit rate for both prefetched data and cached data.

We need to divide the traditional meaning of cache hit into prefetch hit and the narrow interpretation of cache hit. The **prefetch hit** is a read request on a prefetched data that has not yet been referenced by the host since it was prefetched. The **cache hit** is defined as a read request on a cached data that was loaded by the host's past request.

With the absence of efficient cache management for prefetched data, the low prediction accuracy of strip prefetching exhausts the cache memory. To resolve this problem, ASP early evicts (culls) prefetched data that is less valuable than the least-valuable cached data in an adaptive manner with differential feedback so that *the sum of the cache hit rate and the prefetch hit rate of ASP is maximized in a given cache management scheme*. The differential feedback has similar features with the adaptive scheme based on marginal utility used in the sequential prefetching in adaptive replacement cache (SARC) [11]. However, there are several differences between SARC and our scheme, which are discussed at the end of Section 2.3.

Several approaches for balancing the amount of memory given to cached data and prefetched data were taken by Patterson et al.'s transparent informed prefetching [35] and its extension [42]. However, there are many significant differences between them and our scheme. Section 1.2.5 addresses these in more detail.

ASP does not decide which cached data should be evicted and most cache replacement schemes do not take the prefetch hit into account. Hence, ASP may perform well with any of the recent cache replacement policies including RACE [48], ARC [29], SARC [11], AMP [10], and PROMOTE [9].

1.1.5 Prefetching Cost

As well as requiring prefetched data management, prefetching must spend less time in reading the requested data from disks. Hence, ASP include a simple on-line disk simulation to estimate whether strip prefetching requires a larger read cost than no prefetching for the current workload. ASP activates or deactivates strip prefetching depending on this comparison.

If a workload exhibits neither prefetch hits nor cache hits, it is apparent that strip prefetching has a greater cost than no prefetching. In this case, any prefetching should be deactivated, even though ASP efficiently culls uselessly prefetched data. For real workloads, ASP exploits an online disk simulation to estimate the two read costs.

1.2 Related Work

1.2.1 History-Based Prefetching

History-based prefetching, which predicts future accesses by learning the stationary past accesses, has been proposed in various forms. Palmer and Zdonik proposed an associated memory that recognizes access patterns by repeated training [33]. Grimsrud et al. provided an adaptive table, in which an entry is associated with each cluster (of one or more disk blocks) on the disk; furthermore, each entry contains the next cluster for the best prefetching candidate and weight [14]. Griffioen and Appleton suggested a file-level prediction that prefetches files early based on the past file activity [13]. Prefetching with a Markov predictor, which is based on the transition frequency of reference strings, has also been studied [18]. Recording and analyzing past accesses requires a significant amount of memory; hence, several studies have been proposed to reduce the required amount of memory for their history-based prefetching [23, 25, 44].

History-based prefetching, which records and mines the extensive history of past accesses, is cumbersome and expensive to maintain in practical systems. It also suffers from low predictive accuracy, and the resultant unnecessary reads can degrade performance. Furthermore, it is effective only for stationary workloads.

1.2.2 Application-Hint-Based Prefetching

When a small number of processes or a single process generates a non-sequential access, a small number of concurrent I/Os may not fully exploit disk parallelism in the disk array. In order to solve this problem, Patterson et al. suggested a disclosure hint interface [35]. This interface must be exploited by an application programmer so that information about future accesses can be given through an I/O-control (ioctl) system call. The state-of-art interface, the asynchronous I/O of Linux 2.6 [4], may replace the ioctl system call. The disclosure hint forces programmers to modify applications so that the applications issue hints. Some applications involve significant code restructuring to include disclosure hints.

Speculative execution provides application hints without modifying the code [6]. A copy of the original thread is speculatively executed without affecting the original execution in order to predict future accesses. However, the speculative thread consumes considerable computational resources and can cause misprediction if the future accesses rely on the data of past accesses.

1.2.3 Offline Optimal Prefetching

Traditional buffer management algorithms that minimize cache misses are substantially suboptimal in parallel I/O systems where multiple I/Os can proceed simultaneously [19]. Analytically optimal prefetching and caching

schemes have been studied with respect to situations in which future accesses are given [19, 22]. These schemes are optimal in terms of cache hit rate and disk parallelism. As a metric value, however, the cache hit rate may not accurately reflect the real performance because a sequential read for tens of blocks can achieve a much higher disk throughput than random reads for two blocks [8]. Furthermore, offline prefetching does not resolve the two conflicting problems of parallelism loss and independency loss.

1.2.4 Sequential Prefetching

The most common form of prefetching is sequential prefetching (SEQP), which is widely used in a variety of operating systems because sequential accesses are common in practical systems.

The *Atropos* volume manager [36] dramatically reduces disk positioning time for sequential accesses to two dimensional data structures by means of new data placement in disk arrays. However, *Atropos* does not give no solution at all to the five problems addressed in Introduction.

Table-based prefetching (TaP) [26] detects these sequential patterns in a storage cache without any help of file systems, and dynamically adjusts the cache size for sequentially prefetched data, namely prefetch cache size, which is adjusted to an efficient size that obtains no more prefetch hit rate above the preset level even if the prefetch cache size is increased. However, TaP fails to consider both prefetch hit rate and cache hit rate.

MSP, which was proposed in our earlier work [2], detects semi-sequential reads at the block level. A sequential read at the file level exhibits a *semi-sequential* read at the block level because reads at the block level require both metadata access and data access throughout in different regions and the file may be fragmented. If a long semi-sequentiality is detected, the prefetch size of MSP becomes a multiple of the stripe size, and the prefetching request of MSP is aligned in the stripe; as a result, MSP achieves perfect disk parallelism.

Although sequential prefetching is the most popular prefetching scheme in practical systems, sequential prefetching and its variations have until now failed to consider independency loss and they are not at all beneficial to non-sequential reads.

1.2.5 Prefetching and Caching

Among offline prefetching schemes, Kallahalla [19] and Kimbrel [22] took both prefetched data and cached data into account in order to increase cache hit rate. However, their approaches are not realizable in actual systems since they require complete knowledge of future accesses. Patterson et al. [35, 42] provided practical schemes known as TIP and TIPTOE (TIP with Temporal

Overload Estimators, an extension of TIP), TIP and TIPTOE estimate the read service time of prefetched data, the hinted cache, and the shrinkage of the cached data. They then choose the globally least-costly block cache for the victim of eviction.

Distinguishing prefetched data from cached data is the common part of TIP, TIPTOE, and our ASP. However, there are notable differences between the first two and ASP. (1) While their eviction policy does not manage uselessly prefetched data, ASP can manage inaccurately prefetched data that are prestaged by strip prefetching. This provides noticeable benefits if the workload exhibits spatial locality. (2) They are based on an approximate I/O service time model to assess the costs for each prefetched block. This deterministic cost estimation may be different from the real costs and cause errors, while ASP uses an adaptive manner that measures and utilizes the instantaneous real cost of prefetched data and cached data. (3) They must scan block caches to find the least-valuable one, while ASP has a negligible overhead with $O(1)$ complexity.

2 Adaptive Strip Prefetching

To resolve the five problems mentioned in Introduction, we propose adaptive strip prefetching (ASP), which includes three new schemes, *strip prefetching*, *adaptive cache culling*, and an *online disk simulation*.

Strip prefetching resolves independency loss by dedicating each prefetching request to a single disk, and resolves parallelism loss when combined with MSP [2]. Adaptive cache culling eliminates the inaccurate prediction of strip prefetching by providing an optimal point that improves both cache hit rate and prefetch hit rate. To guarantee no performance degradation, ASP enables or disables strip prefetching by using a simple online disk simulation. The following sections describe the three new components comprising ASP.

2.1 Strip Prefetching

To resolve independency loss and to be beneficial for non-sequential reads as well as sequential reads, we propose strip prefetching. Whenever the block requested by the host is not in the cache, strip prefetching reads all blocks of the strip to which the requested block belongs. By grouping consecutive blocks of each strip into a single prefetch unit, strip caches exploit the principle of spatial locality by implicitly prefetching data that is likely to be referenced in the near future. Because each strip is dedicated to only one disk, each prefetch request is not laid across multiple disks; as a result, the problem of independency loss is resolved.

However, strip prefetching has two drawbacks. First, strip prefetching may degrade memory utilization by prefetching useless data. Hence, we propose adaptive

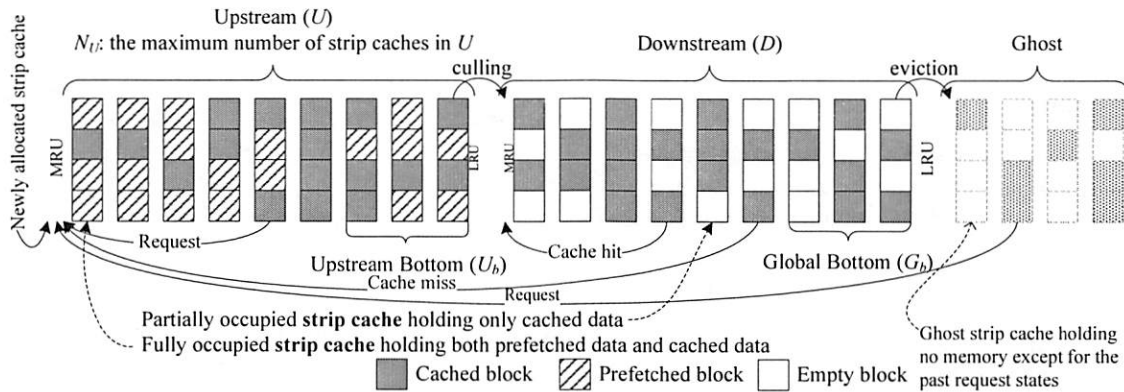


Figure 3: The cache management of ACC: the cache is partitioned into the upstream U and the downstream D . The upstream bottom U_b is a bottom portion of U . The global bottom G_b is a bottom portion of the global list consisting of U and D . ACC changes the variable N_U , the maximum number of strip caches that U can accommodate, in an adaptive manner. If the number of strip caches in U exceeds N_U , ACC evicts (culls) the prefetched block caches of the LRU strip cache of U and moves the strip cache into the MRU position of D .

cache culling to improve strip prefetching. Adaptive cache culling evicts (culls) prefetched and unused data at the proper time in an adaptive manner; as a result, the sum of the cache hit rate and the prefetch hit rate of ASP is guaranteed to be equal to or larger than those of either strip prefetching or no prefetching.

Second, the read service time of strip prefetching may be longer than that of no prefetching. Hence, ASP activates or deactivates strip prefetching by estimating whether the read cost of is less than not prefetching for the current workload. The estimation can be performed by an online disk simulation with low overhead.

2.2 Adaptive Cache Culling

To efficiently manage data prefetched by strip prefetching, as a component of ASP, we propose an adaptive cache culling (ACC) scheme, which maximizes the sum of the cache hit rate and the prefetch hit rate (see Section 1.1.4 for this terminology) of ASP in a given cache management scheme.

Figure 3 illustrates the cache structure that is managed in strip caches, each of which consists of four blocks. ACC manages the cache in strip units. Each strip cache holds the data blocks of a strip. The data block can be a cached block holding cached data, a prefetched block holding prefetched data, or an empty block holding neither memory nor data.

To evict prefetched data earlier than cached data at the proper time in an adaptive manner (described in Section 2.3), as shown in Fig. 3, strip caches are partitioned into the upstream U and the downstream D . U can include both prefetched blocks and cached blocks but D excludes prefetched blocks. Newly allocated strip cache that may hold both prefetched blocks and cached blocks is inserted into U . If the number of strip caches in U exceeds the

maximum number of strip caches that U can accommodate, the LRU strip cache of U moves to D , and ACC **culls** (evicts) the prefetched blocks of this strip cache.

A host request that is delivered to a prefetched block changes it into a cached block. All cached or prefetched blocks make their way from upstream toward downstream like a stream of water. If a prefetched block flows downstream, the prefetched block is changed into an empty block by culling, which deallocates the memory holding the prefetched data but retains the information that the block is empty in D . Thus, blocks requested by the host remain in the cache for a longer time than prefetched and unrequested blocks. The average lifetime of prefetched blocks is determined by the size of U , which is dynamically adjusted by measuring instant hit rates.

ACC changes a variable N_U , the maximum number of strip caches that U can accommodate, in an adaptive manner. If N_U decreases, the prefetch hit rate decreases due to a reduced amount of prefetched data but the cache hit rate increases, otherwise, vice versa. The system performance depends on the total hit rate that is the sum of the prefetched hit rate and the cache hit rate. Section 2.3 describes a control scheme for N_U to maximize the total hit rate, while this section describes the cache management and structure of ACC.

The LRU strip cache of all is evicted under memory pressure. A cache hit for a strip cache of U and D moves the corresponding strip cache to the MRU position of U and D , respectively, like the least recently used (LRU) policy. An evicted strip cache can be a ghost strip cache, which is designed to improve the performance using request history. The ghost strip cache has a loose relationship with the major culling process. Hence, ACC can perform even if we do not manage the ghost strip cache.

The next section describes the ghost strip cache and some issues.

2.2.1 Issues

The stream management of ACC prevents the cache hits in D from moving the hit strip cache to U . If a partially occupied strip cache (including an empty block) of D can move to U , the memory space occupied by U is shrunk because the partially occupied strip cache may evict another fully occupied strip cache from U to keep the number of strip caches of U from being equal to N_U . This process breaks the optimal partition of U and D .

Sibling strip caches are the strip caches corresponding to the strips that belong to the same stripe. Destaging dirty blocks from the RAID cache is more efficient when it is managed in stripes than in blocks or strips [12]. The stripe cache contains sibling strip caches belonging to the stripe. Hence, although a strip cache is evicted, its stripe cache is still alive if at least one sibling strip cache is alive. Therefore, the stripe cache can easily maintain an evicted strip cache as a ghost strip cache.

When all blocks of a strip cache are evicted from the cache, the strip cache becomes a *ghost strip cache* if one of its sibling strip caches is still alive in the cache. Ghost strip caches hold no memory except for request states of the past. The request states indicate which blocks of the strip cache were requested by the host. When an I/O request is delivered for a ghost strip, the ghost strip becomes alive as a strip cache that retains the past request states. This strip cache that was a ghost is called a *reincarnated strip cache*, but which has no distinction with the other strip caches except for the past request states.

The past request states of the ghost strip cache works like a kind of history-based prefetching. The past request states remain even after the ghost strip cache is reincarnated. The past cached block, which had not been requested before the block became a ghost, has high possibility of being referenced by the host in the near future although it has not yet been referenced since it was reincarnated. Thus, *the culling process does not evict past cached blocks as well as cached blocks*.

The cache replacement policy in our implementation is to evict the LRU strip cache of all. However, ACC does not determine a cache replacement policy that may evict any cached block in U or D . A new policy for ACC or a combination of a state-of-the-art cache replacement policy as referenced in Section 1.1.4 and ACC may provide better performance than the LRU scheme. The exact cache replacement policy is not within the scope of this paper.

2.2.2 Summary of Operation

Whenever a request from the host is delivered to the disk array, the two lists, U and D , are managed by the follow-

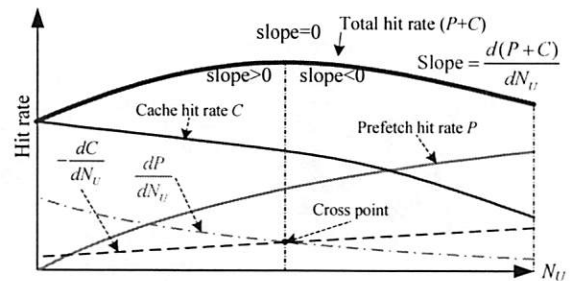


Figure 4: The function of total hit rate ($P + C$) with respect to N_U : when the slope of the function is zero, the total hit rate is at the maximum.

ing rules:

- *A cache hit or prefetch hit occurring in U* : The strip cache that corresponds to the hit moves to the MRU position of U .
- *A cache hit occurring in D* : The strip cache moves to the MRU position of D .
- *A request for a ghost strip cache or an empty block of alive strip caches*: The corresponding strip is read from the disk by controlled strip prefetching (as described in Section 2.4, strip prefetching can be deactivated by the decision algorithm of ASP with an online disk simulation.), and its strip cache is inserted into the MRU position of U .
- *A cache miss on neither alive nor ghost strip caches*: A new strip cache is allocated for the requested block, read by the controlled strip prefetching, and inserted into the MRU position of U .
- *If the number of strip caches exceeds N_U* , the LRU strip cache of U moves to the MRU position of D , and the prefetched blocks except for the past cached blocks are culled (evicted).

2.3 Differential Feedback of ACC

Finding the optimal value of N_U , the maximum number of strip caches occupying U , is the most important part of ACC. We regard N_U as optimal when it maximizes the total hit rate, which is the sum of the prefetch hit rate P and the cache hit rate C . Fig. 4 illustrates a function of the total hit rate with respect to N_U . When the slope of the function is zero, the total hit rate is at the maximum.

As N_U increases, P increases but the incremental rate of P declines. As N_U decreases, C increases and the incremental rate of C declines. Therefore, the derivative of P with respect to N_U , dP/dN_U , is a *monotonically decreasing function* and so is the negative derivative of C with respect to N_U , $-dC/dN_U$. Then, there is **zero or one cross point** of these derivatives. The function of $P + C$ with respect to N_U can form a hill shape or be an monotonically increasing or decreasing function, these three types of functions have only one peak point.

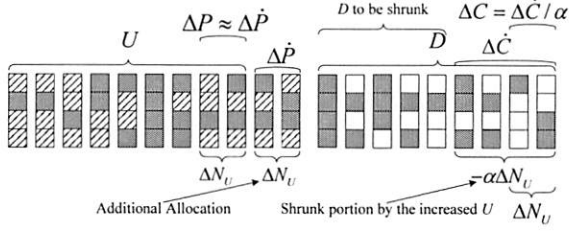


Figure 5: The derivative of P with respect to N_U is similarly equal to the increased number of prefetch hits in U_b when N_U increases slightly. The derivative of C is negatively proportional to the decreased number of cache hits in G_b when the size of U increases.

If we can determine the instant value of the slope for the current N_U , the optimal N_U can be achieved using the following feedback with the instant value of the slope.

$$N_U \leftarrow N_U + C \times \text{slope}, \quad (1)$$

where C is a constant value that determines the speed of the feedback.

When N_U is on the left side of Fig 4, the feedback increases N_U by the positive value of the slope. If N_U increases so much that the slope becomes negative, the feedback with the negative slope decreases N_U so that the total hit rate is near the peak. Even if $P + C$ is an increasing or decreasing function, the above feedback finds the optimal value of N_U .

The function of the slope is the derivative of the total hit rate $P + C$, which is the sum of the two derivatives of P and C with respect to N_U as the following equation.

$$\text{slope} = \frac{d(P + C)}{dN_U} = \frac{dP}{dN_U} + \frac{dC}{dN_U}. \quad (2)$$

An approximated derivative can be measured as shown in Fig. 5. By the definition of differential, the derivative of P is similarly equal to the number of prefetch hits $\Delta \dot{P}$ that, during a time interval, occur in the ΔN_U strip caches additionally allocated to U . The additional prefetch hit rate $\Delta \dot{P}$ is similar to ΔP , which is the prefetch hit rate of the strip caches of the upstream bottom that is adjacent to the additionally allocated strip caches. Therefore, the derivative of the prefetch hit rate with respect to N_U can be approximated with the following equation.

$$\frac{dP}{dN_U} \simeq \frac{\Delta \dot{P}}{\Delta N_U} \simeq \frac{\Delta P}{\Delta N_U}. \quad (3)$$

If the upstream U increases by ΔN_U , the downstream D decreases by $\alpha \Delta N_U$. The coefficient α is determined by the occupancy ratio of the non-empty blocks in both the expanded portion of U and the shrunk portion of D (see Eq. (6)). The derivative of the cache hit rate with

respect to N_U is similar to the cache hit rate $\Delta \dot{C}$ that occurs in the $\alpha \Delta N_U$ strip caches of the global bottom. If we want to monitor the cache hit rate, ΔC , in a fixed size of the global bottom, the derivative of the cache hit rate can be approximated with the following equation.

$$\frac{dC}{dN_U} \simeq -\frac{\Delta \dot{C}}{\Delta N_U} \simeq -\frac{\alpha \Delta C}{\Delta N_U}. \quad (4)$$

The differential value (slope) of the current partition can be obtained by monitoring the number of prefetch hits ΔP occurring in the upstream bottom and the number of cache hits ΔC occurring in the global bottom during a time interval.

$$\text{slope} \propto \Delta P - \alpha \Delta C. \quad (5)$$

The upstream bottom U_b is the bottom portion of U , where the number of strip caches of U_b is predetermined as 20% of the maximum number of strip caches of the simple strip prefetching policy. Similarly, the global bottom G_b is the bottom portion of the global list (the combination of U and D). At an initial stage or in some other cases, D contains no strip cache or too small number of strip caches. Hence, G_b can overlap with U_b . G_b accommodates the same number of strip caches as U_b .

By combining Eqs. (1) and (5), the final feedback operation that achieves the maximum hit rate can be written as in the following process.

$$\alpha = \frac{\text{the number of all blocks in } U_b}{\text{the number of cached blocks in } G_b}. \quad (6)$$

$$N_U \leftarrow N_U + S(\Delta P - \alpha \Delta C). \quad (7)$$

Whenever a hit occurs in U_b or G_b , ACC adjusts N_U using this process. The constant S of Eq. (7) determines the speed of the adaptation. In our experiment, S was speculatively chosen as two. We select the time interval to measure ΔP and ΔC to be the time difference between two successive hits occurring in either U_b or G_b . ΔP and ΔC are hence zero or one.

The upstream size N_U is initially set to the cache size over the strip size. Until the cache is fully filled with data, ASP activates strip prefetching and N_U is not changed by the feedback. At the initial time, D does not exist, and thus, G_b is identical to U_b .

We do not let N_U fall below the size of U_b to retain the fixed size of U_b . If N_U shrinks to the size of U_b , ACC deactivates strip prefetching until N_U swells to twice the size of U_b to make a hysteresis curve. The reason of this deactivation is that too small N_U indicates strip prefetching is not beneficial in terms of not only hit rate but also read service time.

The differential feedback has similar features with the adaptive manner based on marginal utility used in the sequential prefetching in adaptive replacement cache

(SARC) [11]. However, we present a formal method that finds an optimal point using the derivatives of cache hit rate and prefetch hit rate. Only our analytical approach can explain the feedback coefficient α . Furthermore, there are several differences between the two algorithms: (1) while SARC does not distinguish prefetched blocks from cached blocks, our scheme takes care of the eviction of only prefetched blocks that are ignored in SARC, (2) SARC relates to random data and sequential data, whereas the proposed scheme considers both prefetch hits and cache hits in a prefetching scenarios, and (3) our scheme manages the caches in terms of strips for an efficient management of striped disk arrays, while SARC has no feature for disk arrays.

2.4 The Online Cost Estimation

For example, if N_U decreases to the minimum value due to the lack of prefetch hits, ASP must deactivate strip prefetching. The deactivation, however, cannot rely on the minimum size of N_U because N_U may not shrink if both the prefetch hit rate and cache hit rate are zero or extremely low. Therefore, ASP employs an online disk simulation, which investigates whether strip prefetching requires a greater read cost than no prefetching.

Whenever the host requests a block, ASP calculates two read costs, C_n and C_{sp} , by using an online disk simulation with a very low overhead. The disk cost C_{sp} is the virtual read time spent in disks for all the blocks of the cache by pretending that strip prefetching has always been performed. Similarly, the disk cost C_n is the virtual read time spent in disks for all blocks of the current cache by pretending that no prefetching has ever been performed.

Whenever the host requests a block that is not in the cache, ASP compares C_n with C_{sp} . If C_n is less than C_{sp} , ASP deactivates strip prefetching for this request because strip prefetching is estimated to have provided slower read service time than no prefetching for all recent I/Os that have affected the current cache.

If a block hits the cache as a result of strip prefetching, the block must be one of all the blocks in the cache. The gain of strip prefetching is therefore correlated with all blocks in the cache that contains the past data. Hence, all blocks in the cache are exploited to determine whether strip prefetching provides a performance gain.

The cache is managed in terms of strips and consists of N_s strip caches, and each strip cache, S_i , has two variables, $c_n(S_i)$ and $c_{sp}(S_i)$, which are updated by the online disk simulation for every request from the host to the strip cache S_i . The variable $c_n(S_i)$ is the virtual read time spent in S_i if we assume that no prefetching has ever been performed. The variable $c_{sp}(S_i)$ is the virtual read time spent in S_i if we assume that strip prefetching has always been performed. Then we can express C_n and

C_{sp} as follows:

$$C_n \equiv \sum_{i=1}^{N_s} c_n(S_i), \quad C_{sp} \equiv \sum_{i=1}^{N_s} c_{sp}(S_i), \quad (8)$$

where S_i is an i -th strip cache and $\{S_i | 1 \leq i \leq N_s\}$ is the entire cache that consists of N_s strip caches.

From Eq.(8), C_{sp} can be obtained by the sum of $c_{sp}(S_i)$ for all strip caches. It seems to require a great overhead. However, an equivalent operation with $O(1)$ complexity can be achieved as follows: (1) Whenever the host causes a cache miss on a new block whose strip cache is not in U , calculate the disk cost to read all blocks of the requested strip and add it to both $c_{sp}(S_i)$ and C_{sp} ; (2) whenever a read request accesses to a prefetched block or misses the cache, calculate the virtual disk cost to access to the block even though the request hits the cache, and add the disk cost to both C_n and $c_n(S_i)$ of the strip cache S_i to which the requested block belongs; and (3) subtract $c_{sp}(S_i)$ from C_{sp} and subtract $c_n(S_i)$ from C_n when the strip cache S_i is evicted from the cache.

2.5 Combination of ASP and MSP

ASP suffers parallelism loss when there are only a small number of sequential reads. In other words, disks are serialized for a single stream because each request of ASP is dedicated to only one disk. However, the combination of ASP and massive stripe prefetching (MSP), which was briefly described in Section 1.2.4, resolves the parallelism loss of ASP without interfering with the ASP operation and without losing the benefit of ASP.

To combine ASP with MSP, we need the following rules: The reading by MSP does not change any of the cost variables of ASP. When ASP tries to reference a block prefetched by MSP, ASP updates C_{sp} and C_n as if the block has not been prefetched. MSP excludes blocks which are already in the cache or being read or prefetched by ASP or MSP from blocks that are assigned to be prefetched by MSP. ASP culls strip caches prefetched by MSP as well as strip prefetching.

3 Performance Evaluation

3.1 Experimental Setup

We implemented the functions of ASP and MSP in Linux kernel 2.6.18 x86_64 and added them into the RAID driver introduced in our previous work [3], which shows that the RAID driver outperforms the software-based RAID of Linux (MultiDevice) and a hardware-based RAID. We revised our earlier version of the RAID driver for a fine granularity of cache management, and disabled the contiguity transform function of our previous work for all experiments.

The system in the experiments uses dual 3.0 GHz 64-bits Xeon processors, 1 GiB of main memory, two

Adaptec Ultra320 SCSI host bus adapters, and five ST373454LC disks, each of which has a speed of 15,000 revolutions per minute (rpm) and a 75 Gbyte capacity - GiB is the abbreviation of gibibyte, as defined by the International Electrotechnical Commission. 1 GiB refers to 2^{30} bytes, whereas 1 GB refers to 10^9 bytes. [17]. The five disks comprise a RAID-5 array with a strip size of 128 KiB. A Linux kernel (version 2.6.18) for the x86_64 architecture runs on this machine; the kernel also hosts the existing benchmark programs, the ext3 file system, the anticipatory disk scheduler, and our RAID driver, namely, the Layer of RAID Engine (LORE) driver. Apart from the page cache of Linux, the LORE driver has its own cache memory of 500 MiB just as hardware RAID5 have their own cache. The block size is set to 4 KiB.

In our experiments, we compare all possible combinations of our scheme (ASP) and the existing schemes (MSP and SEQP): namely ASP, MSP, SEQP, ASP+MSP, ASP+SEQP, MSP+SEQP, and ASP+MSP+SEQP. The method of combining ASP and MSP is described in Section 2.5. ASP and MSP operate in the LORE driver while SEQP is performed by the virtual file system (VFS) that is independent of the LORE driver. By simply turning on or off the SEQP of VFS, we can easily combine SEQP with ASP or MSP without any rules.

In all the figures of this paper, SEQPX denotes a SEQP with X kibibytes of the maximum prefetch size. For example, SEQP128 indicates that the maximum prefetch size is 128 KiB. All the experimental results are obtained from six repetitions. For some of the results that exhibit significant deviation, each standard deviation is displayed in its graph.

3.2 PCMark: Evaluation of Culling

PCMark®05 records a trace of disk activity during usage of typical applications and bypasses the file system and the operating system's cache [31]. This makes the measurement independent of the file system overhead or the current state of the operating system. Because PCMark is a program for MS-Windows, we recorded the traces of PCMark and replayed the traces three times in Linux with direct IO that makes IOs bypass the operating system's cache.

Figure 6 is the results of the general hard disk drive usage of PCMark that contains disk activities from using several common applications. In this experiment, we compare ASP, strip prefetching (SP), and no prefetching. SP exhibits the highest prefetch hit rate while no prefetching provides the highest cache hit rate. ASP outperforms both SP and no prefetching at all of the cases in Fig. 6. With an over-provisioned memory, the throughput of SP is similar with that of ASP. However, ASP is significantly superior to SP with 400 MiB or less of the RAID cache. At the best case, ASP outperforms SP by 2

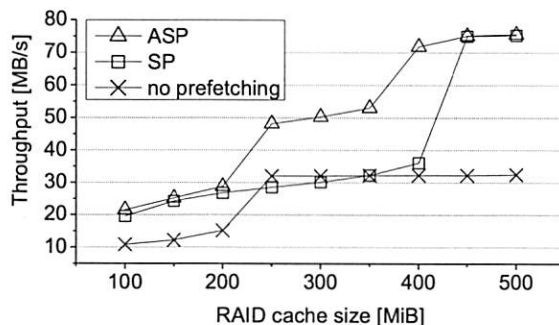


Figure 6: The experimental results of PCMark.

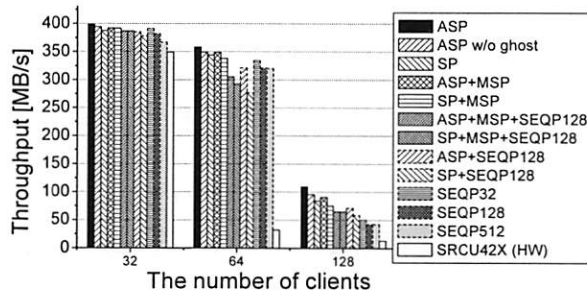


Figure 7: The experimental results of DBench (System Memory : RAID Memory = 512 MiB : 512 MiB). ASP outperforms a hardware-based RAID (Intel SRCU42X) by 11 times for 64 clients

times and no prefetching by 2.2 times with 400 MiB of the RAID cache.

3.3 Dbench

The benchmark Dbench (version 3.04) [43] produces a local file system load. It does all the same read and write calls that the *smbd* server in Samba [37] would produce when confronted with a Netbench [30] run. Dbench generates realistic workloads consisting of so many cache hits, prefetch hits, cache misses, and writes that the ability of ASP can be sufficiently evaluated.

Figure 7 shows the results of Dbench. We divided the main memory of 1 GiB into 512 MiB for the Linux system, 500 MiB for the RAID cache, 12 MiB for the RAID driver. It took 10 minutes for each run.

Both ASP and ASP+MSP rank the highest in Fig. 7 and outperform strip prefetching (SP) by 20% for 128 clients. In this experiment, C_{sp} was always less than C_n ; hence, the gain of ASP over SP originated from ACC. ASP outperforms SEQP32 by 2 times for 128 clients and a hardware-based RAID by 11 times for 64 clients and by 7.6 times for 128 clients. The hardware-based RAID, which is an Intel SRCU42X with a 512 MiB memory, has write-back with a battery pack, cached-IO, and adaptive read-ahead capabilities.

For a fair comparison, the memory of the Linux sys-

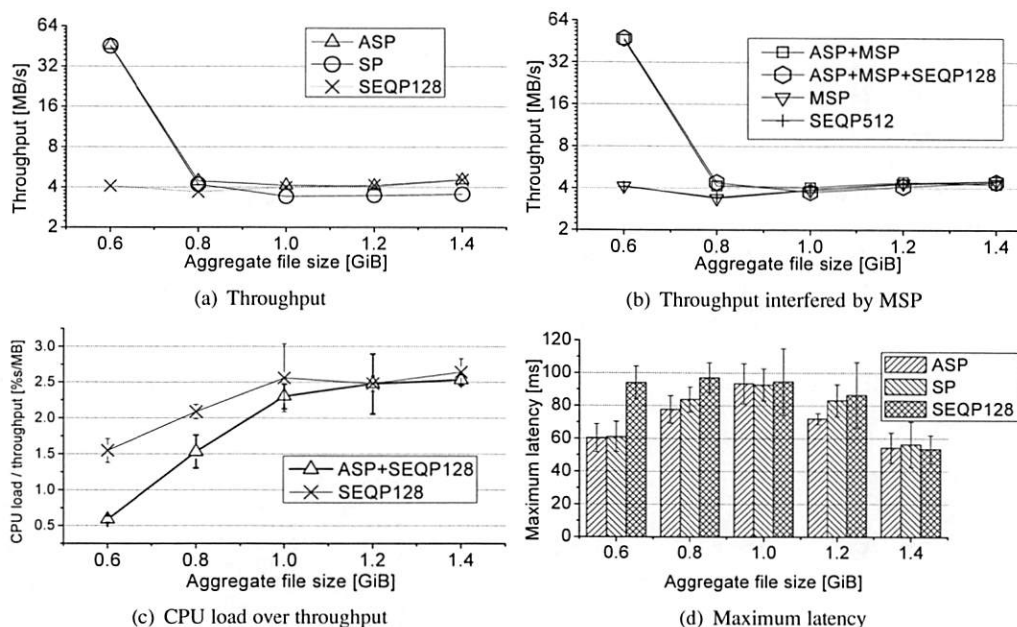


Figure 8: Various metrics with varied aggregate file sizes. Tiobench was used to generate random reads with 4 threads and 40,000 read transactions.

tem for the hardware-based RAID was given the same size (512 MiB) as in the case of our RAID driver, and to compensate the memory occupied by the firmware of the hardware RAID, the cache of our RAID driver is tuned to 500 MiB that is slightly smaller than the internal memory of the hardware RAID. The sequential prefetching of Linux is turned on for the hardware RAID.

Sequential prefetching (SEQP) itself and combinations with SEQP significantly degrade throughput by causing independency loss. Furthermore, because redundant prefetched data that exist in both the RAID cache with ASP and the page cache of Linux with SEQP inefficiently consume cache memory. The addition of SEQP to ASP is inferior to ASP due to independency loss and redundant prefetched data. In addition, the adaptive culling operates inefficiently because prefetched data that are requested by SEQP of the host are considered cached data from the viewpoint of the RAID cache with ASP. Although ASP+SEQP128 is inferior to ASP, it outperforms SEQP128 by 67% for 128 clients.

SEQP with a large prefetch size suffers from cache wastage for a large number of processes. As shown in Fig. 7, SEQP32 outperforms SEQP512 by 4% for 64 clients and by 15% for 128 clients. In contrast, Fig. 9(a) shows that an increase in the maximum prefetch size of SEQP improves disk parallelism for a small number of processes.

“ASP w/o ghost” in Fig. 7 indicates the improved culling effect with the past cached block. ASP outperforms ASP without the ghost by 14% for 128 clients. The

addition of MSP to ASP slightly degrades ASP. However, the addition of MSP can resolve parallelism loss for small numbers of processes or a single process. Section 3.5 shows the evaluation of parallelism loss.

3.4 Tiobench: Decision Correctness

We used the benchmark Tiobench (version 0.3.3) [24] to evaluate whether the disk simulation of ASP makes the right decision on the basis of the workload property. Figure 8(a) shows the throughput of random reads with four threads, 40,000 read transactions, and 4 KiB of the block size in relation to a varied aggregate file size (workspace size). A random read with a small workspace size of 0.6 GiB exhibits such high spatial locality that ASP outperforms SEQP128. With a workspace size of 0.8 GiB, ASP outperforms both SP and SEQP. When the workspace size is equal to or larger than 1 GiB, ASP outperforms SP and show the same throughput as SEQP by deactivating SP. For random reads, SEQP does not prefetch anything. This random read with the large workspace generates so few cache hits that the adaptive cache culling does not improve the throughput of ASP.

Figure 8(b) shows that when combined with other prefetching schemes MSP is effectively disabled without interfering with the combined prefetching scheme for random reads.

Figure 8(c) shows the CPU load over throughput and the standard deviation for the random reads. Although ASP+SEQP128 requires a greater computational overhead than SEQP128, Fig. 8(c) shows that the CPU load

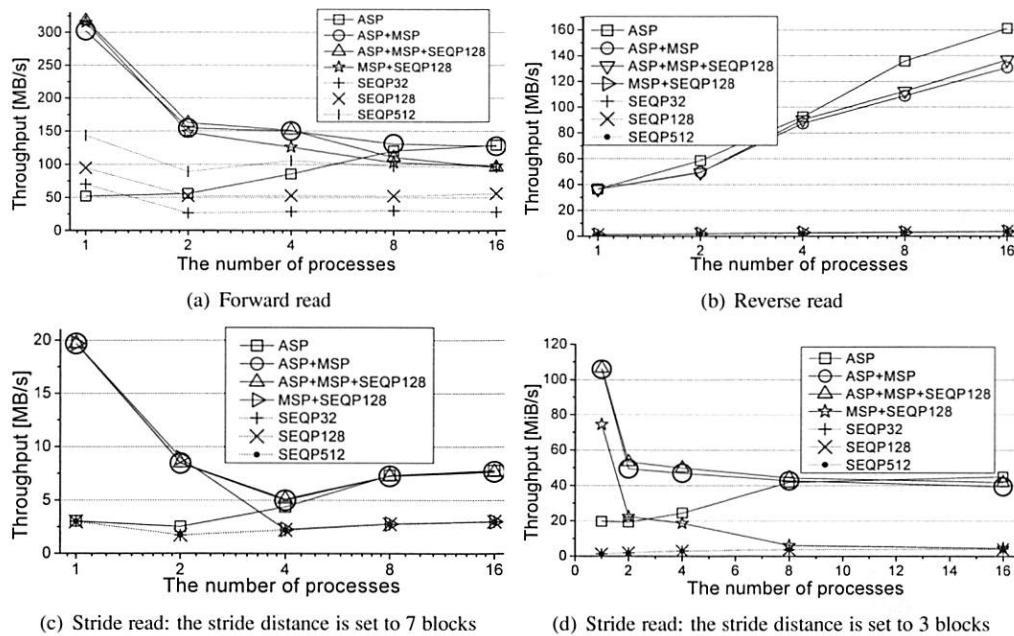


Figure 9: Throughput obtained from the benchmark IOzone for three types of concurrent sequential reads: the forward read, the reverse read, and the stride read. The Throughput is shown in relation to the number of processes, ranging from 1 to 16, with a fixed aggregate file size of 2 GiB and a block size of 4 KiB.

over throughput of ASP+SEQP128 is less than or almost equal to that of SEQP128. Thus, the computational overhead of ASP is too low to be measured.

In spite of its throughput gain, SP may cause the latency to increase because it reads more blocks than no prefetching. However, the performance enhancement by the gain of SP decreases the latency. Figure 8(d) shows that the maximum latency of SEQP128 is longer than that of SP and ASP when SP is beneficial.

This section evaluates the decision correctness and overhead of ASP for the two types of workloads, which may or may not be beneficial to SP. However, because many types of reads exhibit spatial locality, many realistic workloads are beneficial to SP. The next sections show the performance evaluation for various workloads that exhibit spatial locality.

3.5 IOzone: Parallelism and Independency

We used the benchmark IOzone (version 3.283) [32] for the three types of concurrent sequential reads: the forward read, the reverse read, and the stride read. We vary the number of processes from one to 16 with a fixed aggregate file size of 2 GiB and a block size of 4 KiB.

As shown in Fig. 9(a), MSP boosts and dominates the forward sequential throughput when the number of processes is two or four as well as one. The combination of ASP and MSP (ASP+MSP) shows 5.8 times better throughput than ASP for a single stream because ASP

suffers the parallelism loss. When the number of processes is four, ASP+MSP also outperforms ASP by 76%.

When the number of process is one, ASP+MSP outperforms SEQP512 by 134% and SEQP32 by 379%. SEQP, If sequential prefetching has the larger prefetch size, it may resolve the parallelism loss for a small number of streams. However, this approach causes cache wastage and independency loss for concurrent multiple sequential reads. When the number of streams is 16, ASP+MSP outperforms ASP+MSP+SEQP128 by 33% because a combination with SEQP gives rise to independency loss.

For the reverse sequential reads shown in Fig. 9(b), MSP and SEQP do not prefetch any block and, as a consequence, severely degrade the system throughput. ASP is the best for the reserve reads and outperforms SEQP by 41 times for 16 processes.

Figure 9(c) and 9(d) show the throughput results of the stride reads; these results were obtained with IOzone. A stride read means a sequentially discontinuous read with a fixed stride distance that is defined by the number of blocks between discontinuous reads. Stride reads cause additional revolutions in some disk models, thereby degrading the throughput. This problem was unveiled in [3]. The stride reads shown in Fig. 9 are significantly beneficial to MSP and ASP because they prefetch contiguous blocks regardless of stride requests.

Figures 9(c) and 9(d) show that as the stride distance

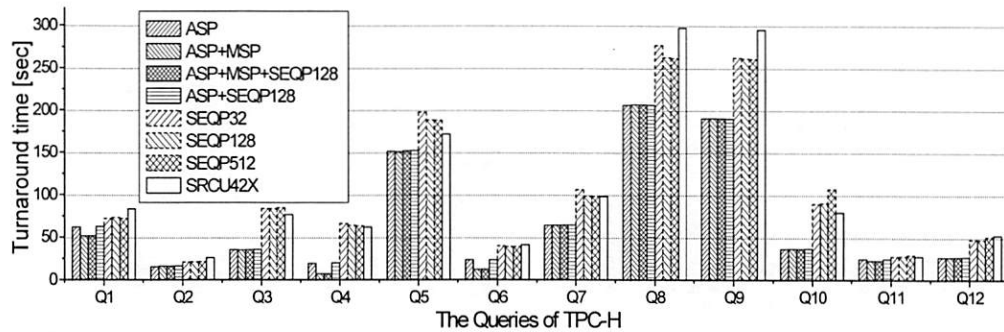


Figure 10: The turnaround time for the queries of TPC-H running on a MySQL database system.

decreases, the throughput of the combinations with ASP increases, whereas the throughput of SEQP decreases. ASP+MSP outperforms any of SEQPs by 84.6 times for a single process when the stride distance is three blocks. As the number of processes increases, MSP is deactivated and ASP dominates the throughput of the stride reads. In Fig. 9(d), the combinations with ASP outperform the combinations without ASP by at least 9.44 times when the number of processes is 16.

3.6 The TPC BenchmarkTMH

The TPC BenchmarkTMH (TPC-H) is a decision support benchmark that runs a suite of business-oriented ad-hoc queries and concurrent data modifications on a database system. Figure 10 shows the results of the queries of TPC-H running on a MySQL database system with the scale factor of one.

Most read patterns of TPC-H are stride reads and non-sequential reads with spatial locality [48], which are highly beneficial to ASP but not to SEQP. ASP and ASP+MSP outperform SEQP128 by 1.7 times and 2.2 times, respectively, on average for the 12 queries. ASP+MSP and ASP+MSP+SEQP are the best for all the 12 queries of TCP-H. Query four (Q4) delivers the best performance of ASP+MSP with 8.1 times better throughput than that of SEQP128.

The MySQL system with TPC-H exhibits neither independency loss nor enough cache hits at the global bottom. Hence, most of ASP's superior performance originates from the principle of the spatial locality of SP. In the experiments in the next section, we do not compare ASP with SP for the same reason.

3.7 Real Scenarios: Cscope, Glimpse, Link, and Booting

This section presents the results of *cscope* [38], *glimpse* [28], *link*, and *booting* as real applications.

The developer's tool *cscope*, which is used for browsing source code, can build the symbol cross-reference of C source files and allow users to browse through the C source files. Figure 11 shows the turnaround time

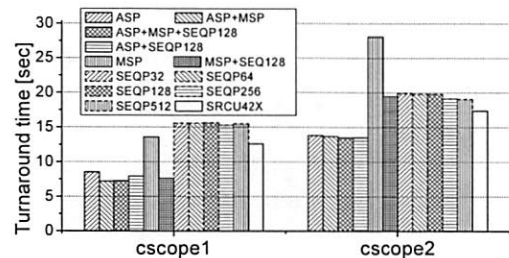


Figure 11: The turnaround time for *cscope* to build the symbol cross-reference of Linux kernel source files.

that is required for *cscope* to index the symbol cross-reference of the Linux kernel (2.6.18) source files. In Fig. 11, *cscope1* means that the symbol indexing is performed without object files that are the results of compilation. The symbol indexing of *cscope2* is performed with source files and object files.

Because *cscope* is an application that uses a single thread, SEQP does not cause independency loss. Hence, ASP+MSP+SEQP128 shows the best performance and ASP+MSP shows the next best performance. However, ASP+MSP+SEQP128 slightly outperforms ASP+MSP by 0.65% for *cscope1* and 1.2% for *cscope2*. In addition, ASP+MSP outperforms SEQP by 2.1 times in *cscope1* and by 38 % in *cscope2*.

In the *cscope2* experiment with both the source files and the object files, the performance gap between ASP+MSP and SEQP decreases. In other words, ASP prefetches unnecessary data because the required source files and the unnecessary object files are interleaved. However, ASP+MSP outperforms SEQP by at least 41%.

Figure 12 shows the execution time needed for *glimpse*, a text information retrieval application from the University of Arizona, to index the text files in "/usr/share/doc" of Fedora Core 5 Release. The results of *glimpse* resemble the results of *cscope1*. ASP+MSP and ASP+MSP+SEQP128 outperform SEQP128 by 106%.

The link of Fig. 12 shows the turnaround time for *gcc* to link the object files of the Linux kernel. The results are obtained by executing "make vmlinux" in the ker-

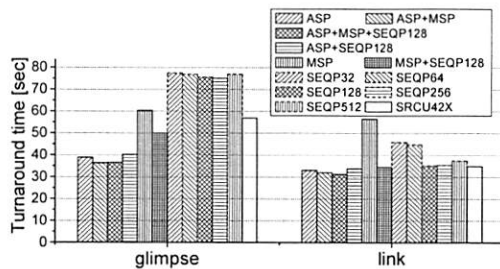


Figure 12: The turnaround time for glimpse to index the text files in “/usr/share/doc” and for gcc to link the object files of the Linux kernel.

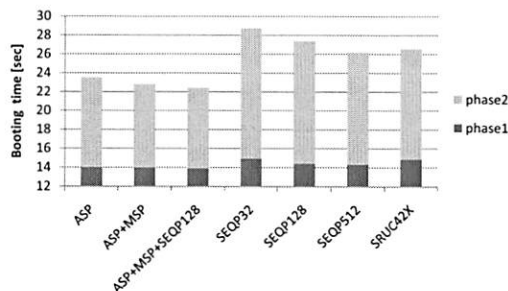


Figure 13: Linux booting speed for the run level 3: Phase 1 is the time that elapses from just before the running of the “init” process to just before the remounting of the root file system with the read-write mode. Phase 2 is the time that elapses from the end of Phase 1 to just after the completion of all the booting scripts (/etc/rc3.d/S* files).

nel source directory after executing “make vmlinux” and “rm vmlinux”. The link operation requires a high computational overhead and small reads that inspect the modification date of the source files. However, ASP+MSP outperforms SEQPs by at least 10%.

Figure 13 shows the Linux booting speed. The Linux booting consists of partially random reads for scattered applications and script files. We partitioned the booting sequence into two phases. Phase 1 shown in Fig. 13 is the time that elapses from just before the running of the “init” process to just before the remounting of the root file system in read-write mode. Phase 2 is the time that elapses from the end of Phase 1 to just after the completion of all the booting scripts (/etc/rc3.d/S* files).

Phase 1 consists of CPU-bounded operations rather than I/O-bounded operations. Hence, the gain of ASP is small. Phase 2, on the other hand, requires I/O bounded operations. Because Linux booting does not produce multiple concurrent reads and consequently causes no independency loss, the best scheme for the Linux booting is ASP+MSP+SEQP128 rather than ASP+MSP. However, the performance gap is negligible. In Phase 2, ASP+MSP outperforms SEQP128 by 46% and SEQP512 by 34%.

4 Conclusion

We introduced five prefetching problems for striped disk arrays in Section 1.1. The five problems are resolved by our scheme as follows: ASP (1) resolves independency loss by aligning the read request in strips that are not laid across disks, (2) resolves parallelism loss by combining with our earlier MSP scheme, which uses the stripe size as the prefetch size to get parallelism only for small numbers of concurrent sequential reads, (3) is beneficial for non-sequential reads as well as sequential reads by exploiting the principle of spatial locality, (4) resolves the inefficient memory utilization of strip prefetching through differential feedback that maximizes the total hit rate in a given cache management scheme, (5) and using an online disk simulation, guarantees less I/O service time than no prefetching.

From the results of Dbench and IOzone, we see that SEQP suffers both parallelism loss and independent loss, but ASP and ASP+MSP are free from them. Additionally, the results of Dbench show that ASP efficiently manages prefetched data. As a result, the sum of the prefetch hit rate and cache hit rate is equal to or greater than that of strip prefetching and no prefetching. The experiments using Tiobench show that ASP has a low overhead and wisely deactivates SP if SP is not beneficial to the current workload. In the results of Dbench, ASP outperforms SEQP128 by 2.3 times and a hardware RAID controller (SRCU42X) by 11 times. The experimental result with PCMark shows that ASP is 2 times faster than SP due to the culling scheme. From the experiments using TPC-H, cscope, link, glimpse, and Linux booting, we can perceive that many realistic workloads exhibit high spatial locality. ASP+MSP is 8.1 times faster than SEQP128 for the query four of TPC-H, and outperforms SEQP by 2.2 times on average for the 12 queries of TPC-H.

Among all the prefetching schemes and combinations presented in this paper, ASP and ASP+MSP rank the highest. We implemented a RAID driver with our schemes in a Linux kernel. Our schemes have a low overhead, and can be used in various RAID systems ranging from entry-level to enterprise-class.

.....+.....
The source code of our RAID driver is downloadable from <http://core.kaist.ac.kr/dn/lore.dist.tgz>, but you may violate some patent rights belonging to KAIST if you commercially use the code.

References

- [1] BAEK, S. H., KIM, B. W., JOUNG, E. J., AND PARK, C. W. Reliability and performance of hierarchical RAID with multiple controllers. In *Proc. of the 20th ACM Symposium on Principles of Distributed Computing* (Aug. 2001), pp. 246–254.
- [2] BAEK, S. H., AND PARK, K. H. Massive stripe cache and prefetching for massive file I/O. In *Proc. of IEEE Int'l Conf. on Consumer Electronics* (Jan. 2007), pp. 5.3–5.
- [3] BAEK, S. H., AND PARK, K. H. Maxtrix-stripe-cache-based contiguity transform for fragmented writes in RAID-5. *IEEE Trans. on Computers* 56, 8 (Aug. 2007), 1040–1054.
- [4] BHATTACHARYA, S., TRAN, J., SULLIVAN, M., AND MASON, C. Linux AIO performance and robustness for enterprise workloads. In *Proc. of Linux Symposium* (July 2004), pp. 63–78.
- [5] BLAUM, M., BRADY, J., AND MENON, J. EVENODD: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Trans. on Computers* 44, 2 (Feb. 1995), 192–201.
- [6] CHANG, F., AND GIBSON, G. A. Automatic I/O hint generation through speculative execution. In *Proc. of the 3rd Symposium on Operating Systems and Design and Implementation* (Feb. 1999), pp. 1–14.
- [7] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys* 26, 2 (June 1994), 145–185.
- [8] DING, X., JIANG, S., AND CHEN, F. A buffer cache management scheme exploiting both temporal and spatial localities. *ACM Trans. on Storage* 3, 2 (June 2007).
- [9] GILL, B. S. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proc. of the 6th USENIX Conference on File and Storage Technologies* (Feb. 2008), pp. 49–65.
- [10] GILL, B. S., AND BATHEN, L. A. D. AMP: Adaptive multi-stream prefetching in a shared cache. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies* (2007).
- [11] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in adaptive replacement cache. In *Proc. of USENIX Annual Technical Conference* (Dec. 2005), pp. 293–308.
- [12] GILL, B. S., AND MODHA, D. S. WOW: Wise ordering for writes = combining spatial and temporal locality in non-volatile caches. In *Proc. of USENIX Conf. File and Storage Technologies* (2005), pp. 129–142.
- [13] GRIFFIOEN, J., AND APPLETON, R. Reducing file system latency using a predictive approach. In *Proc. of USENIX Summer Technical Conf.* (June 1994), pp. 197–208.
- [14] GRIMSRUD, K. S., ARCHIBALD, J. K., AND NELSON, B. E. Multiple prefetch adaptive disk caching. *IEEE Trans. on Knowledge and Data Engineering* 5, 1 (Feb. 1993), 88–103.
- [15] HUANG, X.-M., LIN, C.-R., AND CHEN, M.-S. Design and performance study of rate staggering storage for scalable video in a disk-array-based video server. *IEEE Trans. on Consumer Electronics* 50, 4 (Nov. 2004), 1119–1129.
- [16] HWANG, K., JIN, H., AND HO, R. S. Orthogonal striping and mirroring in distributed RAID for I/O-centric cluster computing. *IEEE Trans. on Parallel and Distributed Systems* 13, 1 (Jan. 2002), 26–44.
- [17] IEC. Prefixes for binary multiples.
- [18] JOSEPH, D., AND GRUNWALD, D. Prefetching using markov predictors. *IEEE Trans. on Computers* 48, 2 (Feb. 1999), 121–133.
- [19] KALLAHALLA, M., AND VARMAN, P. J. PC-OPT: Optimal offline prefetching and caching for parallel I/O systems. *IEEE Trans. on Computers* 51, 11 (Nov. 2002), 1333–1344.
- [20] KENCHAMMANA-HOSEKOTE, D., HE, D., AND HAFNER, J. L. REO: A generic RAID engine and optimizer. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies* (2007).
- [21] KIM, S. H., ZHU, H., AND ZIMMERMANN, R. Zoned-RAID. *ACM Trans. on Storage* 3, 1 (Mar. 2007).
- [22] KIMBREL, T., TOMKINS, A., PATTERSON, R., BERSHAD, B., CAO, P., FELTEN, E., GIBSON, G., KARLIN, A., AND LI, K. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation* (Oct. 1996).
- [23] KROEGER, T. M., AND LONG, D. D. E. Design and implementation of a predictive file prefetching algorithm. In *Proc. of the 2001 USENIX Annual Technical Conference* (June 2001), pp. 105–118.
- [24] KUOPPALA, M. Threaded I/O bench for Linux, 2002.
- [25] LEI, H., AND DUCHAMP, D. An analytical approach to file prefetching. In *Proc. of the 1997 USENIX Annual Technical Conference* (Jan. 1997).
- [26] LI, M., VARKI, E., BHATIA, S., AND MERCHANT, A. TaP: Table-based prefetching for storage caches. In *Proc. of the 6th USENIX Conference on File and Storage Technologies* (Feb. 2008), pp. 81–96.
- [27] LIM, S.-H., JEONG, Y.-W., AND PARK, K. H. Interactive media server with media synchronized raid storage system (nossdav). In *Proc. of Int'l Workshop on Network and Operating System Support for Digital Audio Video* (June 2005), pp. 177–182.
- [28] MANBER, U., AND WU, S. GLIMPSE: A tool to search through entire file systems. In *Proc. of USENIX Winter 1994 Technical Conference* (San Francisco, CA, USA, 1994), pp. 23–32.
- [29] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proc. of USENIX Conf. on File and Storage Technologies* (Mar. 2003), pp. 115–130.
- [30] MEMIK, G., MANGIONE-SMITH, W. H., AND HU, W. NetBench: A benchmarking suite for network processors. In *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD'01)* (2001), pp. 39–43.
- [31] NIEMELÄ, S. *PCMark®05 PC Performance Analysis (white paper)*. Futuremark Corporation, June 2005.
- [32] NORCUTT, W. The IOzone filesystem benchmark, 2007.
- [33] PALMER, M., AND ZDONIK, S. B. Fido: A cache that learns to fetch. In *Proc. of the 17th International Conf. on Very Large Data Bases* (Sept. 1991), pp. 255–264.
- [34] PARK, C.-I. Efficient placement of parity and data to tolerate two disk failures in disk array systems. *IEEE Trans. on Parallel and Distributed Systems* 6, 11 (Nov. 1995), 1177–1184.
- [35] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. of the 15th Symp. on Operating System Principles* (Dec. 1995), pp. 79–95.
- [36] SCHINDLER, J., SCHOLSSER, S. W., SHAO, M., AILAMAKI, A., AND GANGER, G. R. Atropos: A disk array volume manager for orchestrated use of disks. In *Proc. of the 6th USENIX Conference on File and Storage Technologies* (Mar. 2004), pp. 159–172.
- [37] SHARPE, R. Just what is SMB?, 2002.
- [38] STEFFEN, J. L. Interactive examination of a C program with cscope. In *Proc. of USENIX Winter 1985 Technical Conference* (1985), pp. 170–175.
- [39] THE RAID ADVISORY BOARD. *The RAIDBook: A Source Book for RAID Technology sixth edition*. Lino Lakes MN, 1999.
- [40] THOMASIAN, A. Multilevel RAID disk arrays. In *Proc. of the 23rd IEEE/14th NASA Goddard Conf. on Mass Storage Systems and Technologies* (May 2006).
- [41] TIAN, L., FENG, D., JIANG, H., ZHOU, K., ZENG, L., CHEN, J., WANG, Z., AND SONG, Z. PROC: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies* (2007).
- [42] TOMKINS, A., PATTERSON, R. H., AND GIBSON, G. Informed multi-process prefetching and caching. In *Proc. of the 1997 ACM Int'l Conf. on Measurement and Modeling of Computer Systems* (June 1997), pp. 100–114.
- [43] VIEIRA, M., AND MADEIRA, H. A dependability benchmark for OLTP application environments. In *Proc. of the 29th Int'l. Conf. on Very Large Data Bases* (2003).
- [44] VITTER, J. S., AND KRISHNAN, P. Optimal prefetching via data compression. *Journal of the ACM* 43, 5 (Sept. 1996), 771–793.
- [45] WEDDLE, C., OLDHAM, M., QIAN, J., WANG, A.-I. A., AND KUENNING, G. PARAD: A gear-shifting power-aware RAID. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies* (2007).
- [46] WIKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Trans. on Computer Systems* 14, 1 (Feb. 1996), 108–136.
- [47] ZHANG, G., SHU, J., XUE, W., AND ZHENG, W. SLAS: An efficient approach to scaling round-robin striped volumes. *ACM Trans. on Storage* 3, 1 (Mar. 2007).
- [48] ZHU, Y., AND JIANG, H. RACE: A robust adaptive caching strategy for buffer cache. *IEEE Tran. Computers* 57, 1 (Jan. 2008), 25–40.

Context-Aware Prefetching at the Storage Server

Gokul Soundararajan, Madalin Mihailescu[†], and Cristiana Amza

Department of Electrical and Computer Engineering

Department of Computer Science[†]

University of Toronto

Abstract

In many of today's applications, access to storage constitutes the major cost of processing a user request. Data prefetching has been used to alleviate the storage access latency. Under current prefetching techniques, the storage system prefetches a batch of blocks upon detecting an access pattern. However, the high level of concurrency in today's applications typically leads to interleaved block accesses, which makes detecting an access pattern a very challenging problem. Towards this, we propose and evaluate *QuickMine*, a novel, lightweight and minimally intrusive method for context-aware prefetching. Under *QuickMine*, we capture application contexts, such as a transaction or query, and leverage them for context-aware prediction and improved prefetching effectiveness in the storage cache.

We implement a prototype of our context-aware prefetching algorithm in a storage-area network (SAN) built using Network Block Device (NBD). Our prototype shows that context-aware prefetching clearly outperforms existing context-oblivious prefetching algorithms, resulting in factors of up to 2 improvements in application latency for two e-commerce workloads with repeatable access patterns, TPC-W and RUBiS.

1 Introduction

In many of today's applications, such as, e-commerce, on-line stores, file utilities, photo galleries, etc., access to storage constitutes the major cost of processing a user request. Therefore, recent research has focused on techniques for alleviating the storage access latency through storage caching [14, 23, 29] and prefetching techniques [24, 25, 35, 36, 37]. Many traditional storage prefetching algorithms implement sequential prefetching, where the storage server prefetches a batch of sequential blocks upon detecting a sequential access pattern. Recent algorithms, like *C-Miner** [24, 25], capture

repeatable non-sequential access patterns as well. However, the storage system receives interleaved requests originating from many concurrent application streams. Thus, even if the logical I/O sequence of a particular application translates into physically sequential accesses, and/or the application pattern is highly repeatable, this pattern may be hard to recognize at the storage system. This is the case for concurrent execution of several applications sharing a network attached storage, e.g., as shown in Figure 1, and also for a single application with multiple threads exhibiting different access patterns, e.g., a database application running multiple queries, as also shown in the figure.

We investigate prefetching in storage systems and present a novel caching and prefetching technique that exploits logical application contexts to improve prefetching effectiveness. Our technique employs a context tracking mechanism, as well as a lightweight frequent sequence mining [38] technique. The context tracking mechanism captures application contexts in an *application independent manner*, with minimal instrumentation. These contexts are leveraged by the sequence mining technique for detecting block access patterns.

In our context tracking mechanism, we simply tag each application I/O block request with a context identifier corresponding to the higher level application context, e.g., a web interaction, database transaction, application thread, or database query, where the I/O request to the storage manager occurs. Such contexts are readily available in any application and can be easily captured. We then pass this context identifier along with each read block request, through the operating system, to the storage server. This allows the storage server to correlate the block accesses that it sees into frequent block sequences according to their higher level context. Based on the derived block correlations, the storage cache manager then issues block prefetches per context rather than globally.

At the storage server, correlating block accesses is performed by the frequent sequence mining component of

our approach. In particular, we design and implement a lightweight and dynamic frequent sequence mining technique, called *QuickMine*.

Just like state-of-the-art prefetching algorithms [24, 25], *QuickMine* detects sequential as well as non-sequential correlations using a history-based mechanism. *QuickMine*'s key novelty lies in detecting and leveraging block correlations within logical application contexts. In addition, *QuickMine* generates and adapts block correlations *incrementally, on-the-fly*, through a lightweight mining algorithm. As we will show in our experimental evaluation, these novel features make *QuickMine* uniquely suitable for on-line pattern mining and prefetching by i) substantially reducing the footprint of the block correlations it generates, ii) improving the likelihood that the block correlations maintained will lead to accurate prefetches and iii) providing flexibility to dynamic changes in the application pattern, and concurrency degree.

We implement *QuickMine* in a lightweight storage cache prototype embedded into the Network Block Device (NBD) code. We also implement several alternative approaches for comparison with our scheme, including a baseline LRU cache replacement algorithm with no prefetching, and the following state-of-the-art context-oblivious prefetching schemes: two adaptive sequential prefetching schemes [10, 16] and the recently proposed *C-Miner** storage prefetching algorithm [24, 25].

In our experimental evaluation, we use three standard database applications: the TPC-W e-commerce benchmark [1], the RUBiS auctions benchmark and DBT-2 [40], a TPC-C-like benchmark [30]. The applications have a wide range of access patterns. TPC-W and RUBiS are read-intensive workloads with highly repeatable access patterns; they contain 80% and 85% read-only transactions, respectively, in their workload mix. In contrast, DBT-2 is a write-intensive application with rapidly changing access patterns; it contains only 4% read-only transactions in its workload mix. We instrument the MySQL/InnoDB database engine to track the contexts of interest. We found that changing the DBMS to incorporate the context into an I/O request was trivial; the DBMS already tracks various contexts, such as database thread, transaction or query, and these contexts are easily obtained for each I/O operation. We perform experiments using our storage cache deployed within NBD, running in a storage area network (SAN) environment.

Our experiments show that the context-aware *QuickMine* brings substantial latency reductions of up to factors of 2.0. The latency reductions correspond to reductions of miss rates in the storage cache of up to 60%. In contrast, the context oblivious schemes perform poorly for all benchmarks, with latencies comparable to, or worse than the baseline. This is due to either i) inaccurate

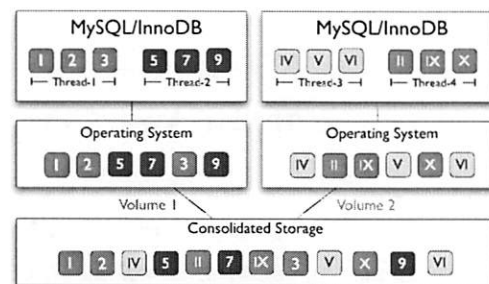


Figure 1: **Interleaved Accesses.** Accesses from concurrent processes/threads are interleaved at the storage server.

prefetches or ii) non-repeatable (false) block correlations at context boundaries, hence useless prefetch rules in the context-oblivious approaches. Our evaluation shows that *QuickMine* generates substantially more effective block correlation rules overall, in terms of both the number of prefetches triggered and the prefetching accuracy. We also show that *QuickMine* is capable of adjusting its correlation rules dynamically, without incurring undue overhead for rapidly changing patterns.

The rest of this paper is organized as follows. Section 2 provides the necessary background and motivates our dynamic, context-aware algorithm. Section 3 introduces our *QuickMine* context-aware prefetching solution. Section 4 provides details of our implementation. Section 5 describes our experimental platform, methodology, other approaches in storage cache management that we evaluate in our experiments. Section 6 presents our experimental results. Section 7 discusses related work and Section 8 concludes the paper.

2 Background and Motivation

We focus on improving the cache hit rate at the storage cache in a SAN environment through prefetching. Our techniques are applicable to situations where the working set of storage clients, like a database system or file system, does not fit into the storage cache hierarchy i.e., into the combined caching capabilities of storage client and server. This situation is, and will remain common in the foreseeable future due to the following reasons.

First, while both client and storage server cache sizes are increasing, so do the memory demands of storage applications e.g., very large databases. Second, previous research has shown that access patterns at the storage server cache typically exhibit long reuse distances [7, 26], hence poor cache utilization. Third, due to server consolidation trends towards reducing the costs of management in large data centers, several applica-

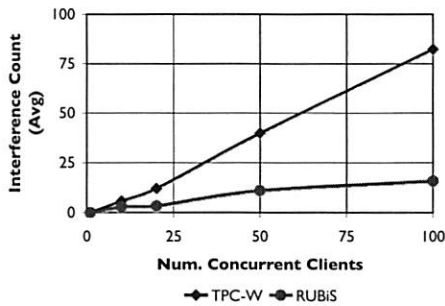


Figure 2: **Interference count.** Access interleavings of different threads with increasing number of concurrent clients.

tions typically run on a cluster with consolidated storage, or even on the same physical server. This creates application interference, hence potential capacity misses, and reduced prefetching effectiveness [19] in the shared storage-level cache.

In the following, we motivate our context-aware prefetching approach through a workload characterization for two e-commerce applications.

2.1 Workload Characterization

We analyze the access patterns of two popular e-commerce benchmarks: TPC-W, and RUBiS. We conduct experiments using a network storage server based on the NBD (network block device) protocol built into Linux. The NBD server manages access to physical storage and provides virtual block devices to applications. We experiment with each benchmark separately, varying the number of clients from 1 to 100. Each application runs on MySQL/InnoDB, which uses the NBD client to mount the virtual block device. We provide more details on our experimental setup in Section 5.

We characterize the access patterns of the two applications using the following metrics. The average/maximum *sequential run length* [39] is the average/maximum length of physically sequential block sequences for the duration of the application's execution. The average *context access length* is the average number of I/O requests issued by a logical unit of work in the application, i.e., by a transaction. Finally, the *interference count* [19] measures the interference in the storage cache, defined as the number of requests from other transactions that occur between consecutive requests from a given transaction stream.

In our experiments, we first compute the sequential run lengths when each thread is run in isolation i.e., on the *de-tangled* trace [39] for each of the two benchmarks. The lengths are: 1.05 (average) and 277 (maximum) for TPC-W and 1.14 (average) and 64 (maximum) for RU-

BiS. We then measure the sequential run lengths on the interleaved access traces, while progressively increasing the number of clients. We find that the sequential run length decreases significantly as we increase the concurrency degree. For example, with 10 concurrently running clients, the sequential run length is already affected: 1.04 (average) and 65 (maximum) for TPC-W, and 1.05 (average) and 64 (maximum) for RUBiS. With the common e-commerce workload of 100 clients, the average sequential run length asymptotically approaches 1.0 for both benchmarks. To further understand the drop in sequential run length, we plot the interference count for each benchmark when increasing the number of clients in Figure 2. The figure shows that the interference count increases steadily with the number of concurrent clients, from 5.87 for TPC-W and 2.91 for RUBiS at 10 clients, to 82.22 for TPC-W and 15.95 for RUBiS with 100 concurrently running clients.

To study the lower interference count in RUBiS compared to TPC-W, we compute the average *context access length* per transaction, in the two benchmarks. We find that the average context access length for RUBiS is 71 blocks, compared to 1223 blocks for TPC-W, 87% of the RUBiS transactions are short, reading only 1 to 10 blocks of data, compared to 79% in TPC-W, and several RUBiS transactions access a single block. Hence in TPC-W, longer logical access sequences result in higher interference opportunities, and for both benchmarks only a few logical sequences translate into physically sequential accesses.

The preliminary results presented in this section show that: i) opportunities for sequential prefetching in e-commerce workloads are low, and ii) random (non-repeatable) access interleavings can occur for the high levels of application concurrency common in e-commerce workloads. Accordingly, these results motivate the need for a prefetching scheme that i) exploits generic (non-sequential) access patterns in the application, and ii) is aware of application concurrency and capable of detecting access patterns per application context. For this purpose, in the following, we introduce the *QuickMine* algorithm that employs data mining principles to discover access correlations at runtime in a context-aware manner.

3 Context-aware Mining and Prefetching

In this section, we describe our approach to context-aware prefetching at the storage server. We first present an overview of our approach, and introduce the terminology we use. We then describe in more detail our technique for tracking application contexts, the *QuickMine* algorithm for discovering block correlations and discuss how we leverage them in our prefetching algorithm.

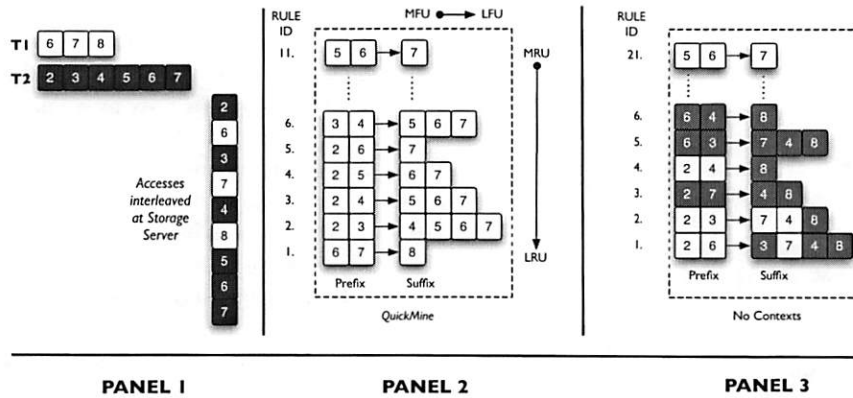


Figure 3: Walk-through. We compare *QuickMine* with a context-oblivious mining algorithm

3.1 Overview

We use application-level contexts to guide I/O block prefetching at the storage server. An application-level *context* is a logical unit of work corresponding to a specific level in the application's structural hierarchy e.g., a thread, a web interaction, a transaction, a query template, or a query instance.

We tag each I/O access with a context identifier provided by the application and pass these identifiers through the operating system to the storage server. This allows the storage server to group block accesses per application-level context. In the example in Figure 1, assuming that the context identifier of an I/O access is the thread identifier, the storage server is able to differentiate that blocks {1, 2, 3} are accessed by *Thread-1* and blocks {5, 7, 9} are accessed by *Thread-2*, from the interleaved access pattern.

Within each sequence of block accesses thus grouped by application context, the storage server applies our frequent sequence mining algorithm, called *QuickMine*. The *QuickMine* algorithm predicts a set of blocks to be accessed with high probability in the near future. The predictions are made based on mining past access patterns. Specifically, *QuickMine* derives *per-context* correlation rules for blocks that appear together frequently for a given context. Creation of new correlation rules and pruning useless old rules for an application and its various contexts occurs incrementally, on-the-fly, while the system performs its regular activities, including running other applications. The *QuickMine* algorithm is embedded in the storage server cache. The storage cache uses the derived rules to issue prefetches for blocks that are expected to occur within short order after a sequence of already seen blocks.

Terminology: For the purposes of our data mining algorithm, a *sequence* is a list of I/O reads issued by an ap-

plication context ordered by the timestamp of their disk requests. A sequence database $D = \{S_1, S_2, \dots, S_n\}$ is a set of sequences. The *support* of a sequence R in the database D is the number of sequences for which R is a *subsequence*. A subsequence is considered *frequent* if it occurs with a frequency higher than a predefined *min-support* threshold. Blocks in frequent subsequences are said to be *correlated*. Correlated blocks do not have to be consecutive. They should occur within a small distance, called a *gap* or *lookahead distance*, denoted G . The larger the *lookahead* distance, the more aggressive the algorithm is in determining correlations. For the purposes of our storage caching and prefetching algorithm, we distinguish between three types of cache accesses. A *cache hit* is an application demand access to a block currently in the storage cache. A *promote* is a block demand access for which a prefetch has been previously issued; the respective block may or may not have arrived at the cache at the time of the demand access. All other accesses are *cache misses*.

3.2 Context Tracking

Contexts are delineated with begin and end delimiters and can be nested. We use our context tracking for database applications and track information about three types of contexts: *application thread*, *database transaction* and *database query*. For database queries, we can track the context of each query instance, or of each query template i.e., the same query with different argument values. We reuse pre-existing begin and end markers, such as, connection establishment/connection tear-down, BEGIN and COMMIT/ROLLBACK statements, and thread creation and destruction to identify the start and end of a context. For *application thread* contexts, we tag block accesses with the thread identifier of the database system thread running the interaction. We differentiate

database transaction contexts by tagging all block accesses between the BEGIN and COMMIT/ABORT with the transaction identifier. A *database query* context simply associates each block access with the query or query template identifier. We studied the feasibility of our tagging approach in three open-source database engines: MySQL, PostgreSQL, and Apache Derby, and we found the necessary changes to be trivial in all these existing code bases. The implementation and results presented in this paper are based on minimal instrumentation of the MySQL/InnoDB database server to track transaction and query template contexts.

While defining meaningful contexts is intuitive, defining the right context granularity for optimizing the prefetching algorithm may be non-trivial. There is a trade-off between using coarse-grained contexts and fine-grained contexts. Fine-grained contexts provide greater prediction accuracy, but may limit prefetching aggressiveness because they contain fewer accesses. Coarse-grained contexts, on the other hand, provide more prefetching opportunities, but lower accuracy due to more variability in access patterns, e.g., due to control flow within a transaction or thread.

3.3 QuickMine

QuickMine derives block correlation rules for each application context as follows. Given a sequence of already accessed blocks $\{a_1, a_2, \dots, a_k\}$, and a lookahead parameter G , *QuickMine* derives block correlation rules of the form $\{a_i \& a_j \rightarrow a_k\}$ for all i, j and k , where $\{a_i, a_j, a_k\}$ is a subsequence and $i < j < k$, $(j - i) < G$ and $(k - i) < G$.

For each rule of the form $\{a \& b \rightarrow c\}$, $\{a \& b\}$ is called a sequence prefix, and represents two blocks already seen on a *hot path* through the data i.e., a path taken repeatedly during the execution of the corresponding application context. For the same rule, $\{c\}$ is one of the block accesses about to follow the *prefix* with high probability and is called a sequence *suffix*. Typically, the same *prefix* has several different suffixes depending on the lookahead distance G and on the variability of the access patterns within the given context, e.g., due to control flow. For each prefix, we maintain a list of possible suffixes, up to a cut-off *max-suffix* number. In addition, with each suffix, we maintain a *frequency* counter to track the number of times the suffix was accessed i.e., the *support* for that block. The list of suffixes is maintained in order of their respective frequencies to help predict the most probable block(s) to be accessed next. For example, assume that *QuickMine* has seen access patterns $\{(a_1, a_2, a_3, a_4), (a_2, a_3, a_4), (a_2, a_3, a_5)\}$ in the past for a given context. *QuickMine* creates rules $\{a_2 \& a_3 \rightarrow a_4\}$ and $\{a_2 \& a_3 \rightarrow a_5\}$ for this context. Further assume that the current access sequence matches

the rule prefix $\{a_2 \& a_3\}$. *QuickMine* predicts that the next block to be accessed will be suffix $\{a_4\}$ or $\{a_5\}$ in this order of probability because $\{a_2, a_3, a_4\}$ occurred twice while $\{a_2, a_3, a_5\}$ occurred only once.

We track the blocks accessed within each context and create/update block correlations for that context whenever a context ends. We maintain all block correlation rules in a *rule cache*, which allows pruning old rules through simple cache replacement policies. The cache replacement policies act in two dimensions: i) for rule prefixes and ii) within the suffixes of each prefix. We keep the most recent *max-prefix* prefixes in the cache. For each prefix, we keep *max-suffix* most probable suffixes. Hence, the cache replacement policies are LRU (Least-Recently-Used) for rule prefixes and LFU (Least-Frequently-Used) for suffixes. Intuitively, these policies match the goals of our mining algorithm well. Since access paths change over time as the underlying data changes, we need to remember recent hot paths and forget past paths. Furthermore, as mentioned before, we need to remember only the most probable suffixes for each prefix. To prevent quick evictions, newly added suffixes are given a grace period.

Example: We show an example of *QuickMine* in Figure 3. We contrast context-aware mining with context-oblivious mining. On the left hand side, we show an example of an interleaved access pattern that is seen at the storage server when two transactions (denoted T_1 and T_2) are running concurrently. Panel 2 shows the result of *QuickMine*. Panel 3 contrasts *QuickMine* with a context-oblivious mining algorithm. To aid the reader, we do not prune/evict items from the rule cache. In Panel 2, we obtain the list of block accesses after T_1 and T_2 finish execution. T_1 accesses $\{6, 7, 8\}$ so we obtain the correlation $\{6 \& 7 \rightarrow 8\}$ (rule 1 in Panel 2). T_2 accesses $\{2, 3, 4, 5, 6, 7\}$ leading to more correlations. We set the lookahead parameter, $G = 5$, so we look ahead by at most 5 entries when discovering correlations. We discover correlations $\{(6 \& 7 \rightarrow 8), (2 \& 3 \rightarrow 4), (2 \& 3 \rightarrow 5), \dots, (5 \& 6 \rightarrow 7)\}$. Finally, we show a snapshot of context-oblivious mining, with false correlations highlighted, in Panel 3. At the end of transaction T_2 , the access pattern $\{2, 6, 3, 7, 4, 8, 5, 6, 7\}$ is extracted and the mined correlations are $\{(2 \& 6 \rightarrow 3), (2 \& 6 \rightarrow 7), (2 \& 6 \rightarrow 4), (2 \& 6 \rightarrow 8), \dots, (5 \& 6 \rightarrow 7)\}$. With context-oblivious mining, false correlations are generated, e.g., $\{(2 \& 6 \rightarrow 3), (2 \& 6 \rightarrow 4), (2 \& 6 \rightarrow 8)\}$ are incorrect. False correlations will be eventually pruned since they occur infrequently, hence have low support, but the pruning process may evict some true correlations as well.

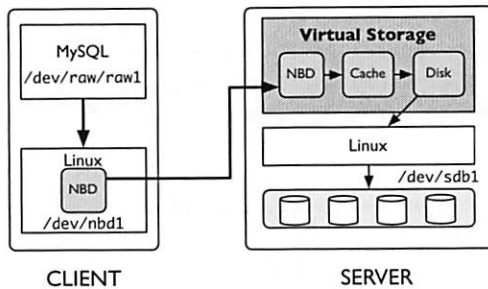


Figure 4: **Storage Architecture:** We show one client connected to a storage server using NBD.

3.4 Prefetching Algorithm

The storage cache uses the block correlation rules to issue block prefetches for predicted future *read* accesses. Block prefetches are issued upon a *read* block miss. We use the last two accesses of the corresponding context to search the rule cache for the prefix just seen in the I/O block sequence. We determine the set of possible blocks to be accessed next as the set of suffixes stored for the corresponding prefix. We prefetch blocks that are not currently in the cache starting with the highest support block up to either the maximum suffixes stored or the maximum prefetching degree.

The number of block prefetches issued upon a block miss called *prefetch aggressiveness*, is a configurable parameter. We set the prefetching aggressiveness to the same value as *max-suffix* for all contexts. However, in heavy load situations, we limit the prefetching aggressiveness to prevent saturating the storage bandwidth. Specifically, we leverage our context information to selectively throttle or disable prefetching for contexts where prefetching is not beneficial. Prefetching benefit is determined per context, as the ratio of prefetches issued versus prefetched blocks used by the application.

The prefetched blocks brought in by one application context may be, however, consumed by a different application context due to data affinity. We call such contexts *symbiotic contexts*. We determine *symbiotic contexts* sets by assigning context identifiers tracking the issuing and using contexts for each prefetched cache block. We then monitor the prefetching benefit at the level of *symbiotic context* sets rather than per individual context. We disable prefetching for contexts (or symbiotic context sets) performing poorly.

4 Implementation Details

We implement our algorithms in our Linux-based virtual storage prototype, which can be deployed over commod-

ity storage firmware. The architecture of our prototype is shown in Figure 4. MySQL communicates to the virtual storage device through standard Linux system calls and drivers, either iSCSI or NBD (network block device), as shown in the figure. Our storage cache is located on the same physical node as the storage controller, which in our case does not have a cache of its own. The storage cache communicates to a *disk module* which maps virtual disk accesses to physical disk accesses. We modified existing *client* and *server* NBD protocol processing modules for the storage client and server, respectively, in order to incorporate context awareness on the I/O communication path.

In the following, we first describe the interfaces and communication between the core modules, then describe the role of each module in more detail.

4.1 Interfaces and Communication

Storage clients, such as MySQL, use NBD for reading and writing logical blocks. For example, as shown in Figure 4, MySQL mounts the NBD device (`/dev/nbd1`) on `/dev/raw/raw1`. The Linux virtual disk driver uses the NBD protocol to communicate with the storage server. In NBD, an I/O request from the client takes the form `<type, offset, length>` where *type* is a *read* or *write*. The I/O request is passed by the OS to the NBD kernel driver on the client, which transfers the request over the network to the NBD protocol module running on the storage server.

4.2 Modules

Each module consists of several threads processing requests. The modules are interconnected through in-memory bounded buffers. The modular design allows us to build many storage configurations by simply connecting different modules together.

Disk module: The disk module sits at the lowest level of the module hierarchy. It provides the interface with the underlying physical disk by translating application I/O requests to the virtual disk into `pread()`/`pwrite()` system calls, reading/writing the underlying physical data. We disable the operating system buffer cache by using direct I/O i.e., the I/O `O_DIRECT` flag in Linux.

Cache module: The cache module supports context-aware caching and prefetching. We developed a portable caching library providing a simple hashtable-like interface modelled after *Berkeley DB*. If the requested block is found in the cache, the access is a cache *hit* and we return the data. Otherwise, the access is a cache *miss*, we fetch the block from the next level in the storage hierarchy, store it in the cache, then return the data. When prefetching is enabled, the cache is partitioned into two

areas: a main cache (MC) and a prefetch cache (PFC). The PFC contains blocks that were fetched from disk by the prefetching algorithm. The MC contains blocks that were requested by application threads. If an application thread requests a block for which a prefetch has been issued, we classify the access as a *promote*. A block *promote* may imply either waiting for the block to arrive from disk, or simply moving the block from PFC to MC if the block is already in the cache.

We use *Berkeley DB* to implement the rule cache and store the mined correlations. The caching keys are rule *prefixes* and the cached data are the rule *suffixes*. The suffixes are maintained using the LFU replacement algorithm and the prefixes are maintained using LRU. The LRU policy is implemented using *timeouts*, where we periodically purge old entries. We configure *Berkeley DB*'s environment to use a memory pool of 4MB.

NBD Protocol module: We modify the original NBD protocol module on the server side, used in Linux for virtual disk access, to convert the NBD packets into our own internal protocol packets, i.e., into calls to our server cache module.

4.3 Code Changes

To allow context awareness, we make minor changes to MySQL, the Linux kernel, and the NBD protocol.

Linux: The changes required in the kernel are straightforward and minimal. In the simplest case, we need to pass a context identifier on I/O calls as a separate argument into the kernel. In order to allow more flexibility in our implementation, and enhancements such as per-context tracking of prefetch effectiveness, we pass a handle to a context structure, which contains the transaction identifier, and query template identifier.

We add three new system calls, `ctx_pread()`, `ctx_pwrite()` and `ctx_action()`. `ctx_action()` allows us to inform the storage server of context begin/end delimiters. Each system call takes a `struct context *` as a parameter representing the context of the I/O call. This context handle is passed along the kernel until it reaches the lowest level where the kernel contacts the block storage device. Specifically, we add a field `context` to `struct request`, which allows us to pass the context information through the I/O subsystem with no additional code changes. Once the I/O request reaches the NBD driver code, we copy the context information into the NBD request packet and pass the information to the storage server.

NBD Protocol: We simply piggyback the context information on the NBD packet. In addition, we add two new messages to the NBD protocol, for the corresponding system call `ctx_action()`, to signify the begin-

ning of a context (`CTX_BEG`) and the end of a context (`CTX_END`).

MySQL: A THD object is allocated for each connection made to MySQL. The THD object contains all contextual information that we communicate to the storage server. For example, `THD.query` contains the query currently being executed by the thread. We generate the query template identifier using the query string. In addition, we call our `ctx_action()` as appropriate, e.g., at transaction *begin/end* and at connection setup/tear-down to inform the storage server of the start/end of a context.

5 Evaluation

In this section, we describe several prefetching algorithms we use for comparison with *QuickMine* and evaluate the performance using three industry-standard benchmarks: TPC-W, RUBiS, and DBT-2.

5.1 Prefetching Algorithms used for Comparison

In this section, we describe several prefetching algorithms that we use for comparison with *QuickMine*. These algorithms are categorized into sequential prefetching schemes (RUN and SEQ) and history based prefetching schemes (*C-Miner**). The RUN and *C-Miner** algorithms share some of the features of *QuickMine*, specifically, some form of concurrency awareness (RUN) and history-based access pattern detection and prefetching (*C-Miner**).

Adaptive Sequential Prefetching (SEQ): We implement an adaptive prefetching scheme following the sequence-based read-ahead algorithm implemented in Linux [10]. Sequence based prefetching is activated when the algorithm detects a sequence (*S*) of accesses to *K* contiguous blocks. The algorithm uses two windows: a *current window* and a *read-ahead window*. Prefetches are issued for blocks contained in both windows. In addition, the number of prefetched blocks used within each window is monitored i.e., the block hits in the current window and the read ahead window, *S.curHit* and *S.reaHit*, respectively. When a block is accessed in the *read-ahead window*, the *current window* is set to the *read-ahead window* and a new read-ahead window of size is $2 * S.curHit$ is created. To limit the prefetching aggressiveness, the size of the read-ahead window is limited by 128KB as suggested by the authors [10].

Run-Based Prefetching (RUN): Hsu et al. [16] show that many workloads, particularly database workloads, do not exhibit strict sequential behavior, mainly due to high application concurrency. To capture sequentiality in a multi-threaded environment, Hsu et al. introduce *run-based prefetching* (RUN) [17]. In *run-based prefetch-*

ing, prefetching is initiated when a sequential run is detected. A reference r to block b is considered to be part of a sequential run R if b lies within $-extent_{backward}$ and $+extent_{forward}$ of the largest block accessed in R , denoted by $R.maxBlock$. This modified definition of sequentiality thus accommodates small jumps and small reverses within an access trace. Once the size of the run R exceeds a sequentiality threshold (32KB), prefetching is initiated for 16 blocks from $R.maxBlock + 1$ to $R.maxBlock + 16$.

History-based Prefetching: There are several history based prefetching algorithms proposed in recent work [17, 13, 20, 26]. *C-Miner** is a static mining algorithm that extracts frequent block subsequences by mining the entire sequence database [25].

*C-Miner** builds on a frequent subsequence mining algorithm, *CloSpan* [42]. It differs from *QuickMine* by mining block correlations off-line, on a long sequence of I/O block accesses. First, *C-Miner** breaks the long sequence trace into smaller sequences and creates a sequence database. From these sequences, as in *QuickMine*, the algorithm considers frequent sequences of blocks that occur within a *gap* window. Given the sequence databases and using the *gap* and *min_support* parameters, the algorithm extracts frequent *closed* sequences, i.e., subsequences whose support value is different from that of its super-sequences. For example, if $\{a_1, a_2, a_3, a_4\}$ is a frequent subsequence with support value of 5 and $\{a_1, a_2, a_3\}$ is a subsequence with support value of 5 then, only $\{a_1, a_2, a_3, a_4\}$ will be used in the final result. On the other hand, if $\{a_1, a_2, a_3\}$ has a support of 6 then, both sequences are recorded. For each *closed* frequent sequence e.g., $\{a_1, a_2, a_3, a_4\}$, *C-Miner** generates association rules of the form $\{(a_1 \rightarrow a_2), (a_1 \& a_2 \rightarrow a_3), \dots, (a_3 \rightarrow a_4)\}$.

As an optimization, *C-Miner** uses the frequency of a rule suffix in the rule set to prune predictions of low probability through a parameter called *min_confidence*. For example, if the mined trace contains 80 sequences with $\{a_2 \& a_3 \rightarrow a_4\}$ and 20 sequences with $\{a_2 \& a_3 \rightarrow a_5\}$, then $\{a_2 \& a_3 \rightarrow a_5\}$ has a (relatively low) confidence of 20% and might be pruned depending on the *min_confidence* threshold. In our experiments, we use *max_gap* = 10, *min_support* = 1, and *min_confidence* = 10% for *C-Miner**.

5.2 Benchmarks

TPC-W¹⁰: The TPC-W benchmark from the Transaction Processing Council [1] is a transactional web benchmark designed for evaluating e-commerce systems. Several web interactions are used to simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer

population. We use 100K items and 2.8 million customers which results in a database of about 4 GB. We use the *shopping* workload that consists of 20% writes. To fully stress our architecture, we create TPC-W¹⁰ by running 10 TPC-W instances in parallel creating a database of 40 GB.

RUBiS¹⁰: We use the RUBiS Auction Benchmark to simulate a bidding workload similar to e-Bay. The benchmark implements the core functionality of an auction site: selling, browsing, and bidding. We do not implement complementary services like instant messaging, or newsgroups. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during the visitor sessions, during a buyer session, users can bid on items and consult a summary of their current bid, rating, and comments left by other users. We are using the default RUBiS bidding workload containing 15% writes, considered the most representative of an auction site workload according to an earlier study of e-Bay workloads [34]. We create a scaled workload, RUBiS¹⁰ by running 10 RUBiS instances in parallel.

DBT-2: DBT-2 is an OLTP workload derived from the TPC-C benchmark [30, 40]. It simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. Each warehouse has 10 sales districts and each district serves 3000 customers. The workload involves transactions from a number of terminal operators centered around an order entry environment. There are 5 main transactions for: (1) entering orders (*New Order*), (2) delivering orders (*Delivery*), (3) recording payments (*Payment*), (4) checking the status of the orders (*Order Status*), and (5) monitoring the level of stock at the warehouses (*Stock Level*). Of the 5 transactions, only *Stock Level* is read only, but constitutes only 4% of the workload mix. We scale DBT-2 by using 256 warehouses, which gives a database footprint of 60GB.

5.3 Evaluation Methodology

We run our Web based applications on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database storage engine. For the database applications, we use the test harness provided by each benchmark while hosting the database on MySQL. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the NBD server. We run the NBD server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. To maximize IO bandwidth, we use RAID 0 on 15

10K RPM 250GB hard disks. We install Ubuntu 6.06 on both the client and server machines with Linux kernel version 2.6.18-smp.

We configure our caching library to use 16KB block size to match the MySQL/InnoDB block size. We use 100 clients for TPC-W¹⁰ and RUBiS¹⁰. For DBT-2, we use 256 warehouses. We run each experiment for two hours. We train *C-Miner** on a trace collected from the first hour of the experiment. We measure statistics for both *C-Miner** and *QuickMine* during the second hour of the experiment. We use a lookahead value of 10 for *C-Miner**, which is the best value found experimentally for the given applications and number of clients used in the experiments. *QuickMine* is less sensitive to the lookahead value, and any lookahead value between 5 and 10 gives similar results. We use a lookahead value of 5 for *QuickMine*.

6 Results

We evaluate the performance of the following schemes: a baseline caching scheme with no prefetching (denoted as LRU), adaptive sequential prefetching (SEQ), run-based prefetching (RUN), *C-Miner** (CMINER), and *QuickMine* (QMINE). In Section 6.1, we present the overall performance of those schemes, whereas in Section 6.2, we provide detailed analysis to further understand the achieved overall performance of each scheme.

6.1 Overall Performance

In this section, we measure the storage cache hit rates, miss rates, promote rates and the average read latency for each of the prefetching schemes by running our three benchmarks in several cache configurations. For all experiments, the MySQL/InnoDB buffer pool is set to 1024MB and we partition the storage cache such that the prefetching area is a fixed 4% of the total storage cache size. For TPC-W¹⁰ and RUBiS¹⁰, we use 100 clients and we vary the storage cache size, showing results for 512MB, 1024MB, and 2048MB storage cache for each benchmark. For DBT-2, we show results only for the 1024MB storage cache, since results for other cache sizes are similar. In *QuickMine*, we use query-template contexts for TPC-W and RUBiS and transaction contexts for DBT-2. However, the results vary only slightly with the context granularity for our three benchmarks.

TPC-W: Figures 5(a)-5(c) show the hit rates (black), miss rates (white), and promote rates (shaded) for all prefetching schemes with TPC-W¹⁰. For a 512MB storage cache, as shown in Fig. 5(a), the baseline (LRU) miss rate is 89%. The sequential prefetching schemes reduce the miss rate by 5% on average. *C-Miner** reduces the

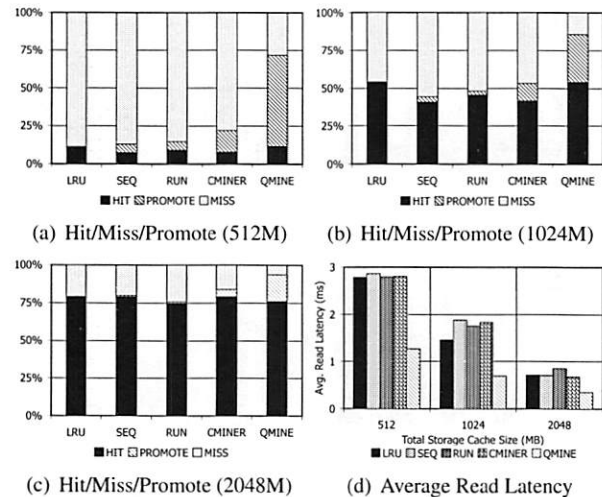


Figure 5: TPC-W. Prefetching benefit in terms of miss rate and average read latency.

miss rate by 15%, while *QuickMine* reduces the miss rate by 60%. The benefit of sequential prefetching schemes is low due to lack of sequentiality in the workload. With larger cache sizes, the baseline miss rates are reduced to 45% for the 1024MB cache (Fig. 5(b)) and to 20% for the 2048MB cache (Fig. 5(c)). With lower miss rates, there are lower opportunities for prefetching. In spite of this, *QuickMine* still provides a 30% and 17% reduction in miss rates for the 1024MB and 2048MB cache sizes, respectively.

For *QuickMine*, the cache miss reductions translate into substantial read latency reductions as well, as shown in Figure 5(d). In their turn, these translate into decreases in overall storage access latency by factors ranging from 2.0 for the 512MB to 1.22 for the 2048MB caches, respectively. This is in contrast to the other prefetching algorithms, where the average read latency is comparable to the baseline average latency for all storage cache sizes.

RUBiS: Context-aware prefetching benefits RUBiS as well, as shown in Figure 6. The baseline (LRU) cache miss rates are 85%, 36% and 2% for the 512MB, 1024MB, and 2048MB cache sizes, respectively. For a 512MB cache, as shown in Figure 6(a), the SEQ and RUN schemes reduce the miss rate by 6% and 9%, respectively. *C-Miner** reduces the miss rate by only 3% while *QuickMine* reduces the miss rate by 48%. In RUBiS, the queries access a spatially-local cluster of blocks. Thus, triggering prefetching for a weakly sequential access pattern, as in RUN, results in more promotes than for SEQ. *C-Miner** performs poorly for RUBiS because many RUBiS contexts are short. This results in many false correlations across context boundaries in *C-Miner**.

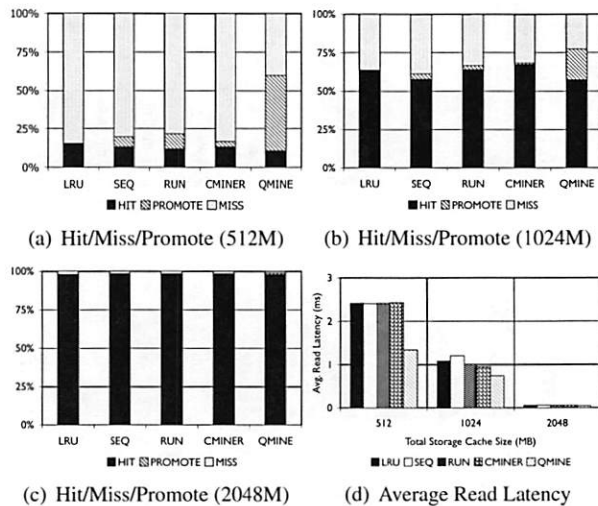


Figure 6: RUBiS. Prefetching benefit in terms of miss rate and average read latency.

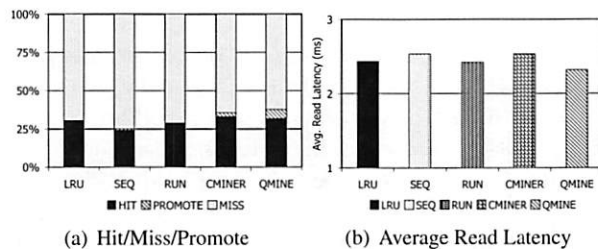


Figure 7: DBT-2. Prefetching benefit in terms of miss rate and average read latency for 1024MB storage cache.

Hence, only a few prefetch rule prefixes derived during training can be later matched on-line, while many rule suffixes are pruned due to low confidence. *QuickMine* overcomes the limitations of *C-Miner** by tracking correlations per context.

The performance of the prefetching algorithms is reflected in the average read latency as well. As shown in Figure 6(d), the sequential prefetching schemes (SEQ and RUN) reduce the average read latency by up to 10% compared to LRU. The reductions in miss rate using *QuickMine* translate to reductions in read latencies of 45% (512MB) and 22% (1024MB) compared to LRU, corresponding to an overall storage access latency reduction by a factor of 1.63 for the 512MB cache and 1.3 for the 1024MB cache.

DBT-2: Prefetching is difficult for DBT-2, since the workload mix for this benchmark contains a high fraction of writes; furthermore, some of its transactions issue very few I/Os. The I/O accesses are not sequential. As Figure 7 shows, the lack of sequentiality causes the sequential prefetching algorithms to perform poorly. Se-

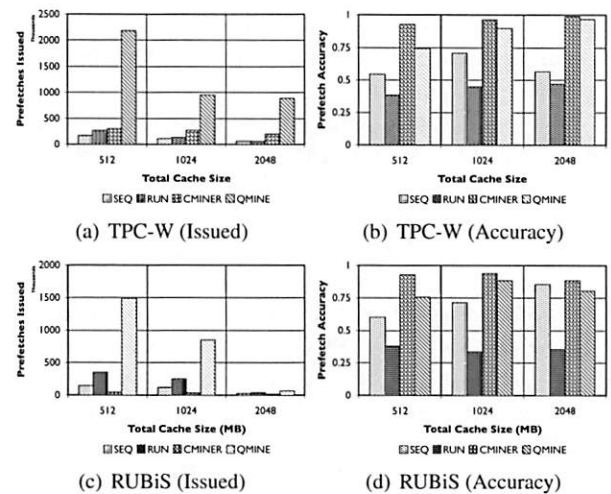


Figure 8: RUBiS¹⁰/TPC-W¹⁰. We show the number of prefetches issued (in thousands) and the accuracy of the prefetches.

quential prefetching schemes decrease the miss rate by less than 1%. *C-Miner** and *QuickMine* perform slightly better. *C-Miner** lowers the miss rate by 2% and *QuickMine* reduces the miss rate by 6%. However, the high I/O footprint of this benchmark causes disk congestion, hence increases the promote latency. Overall, the average read latency increases by 2% for the sequential prefetching schemes and *C-Miner**, while the read latency is reduced by 3% for *QuickMine*.

6.2 Detailed Analysis

In this section, we evaluate the prefetching *effectiveness* of the different schemes by measuring the number of prefetches issued and their *accuracy*, i.e., the percentage of prefetched blocks consumed by the application. Both metrics are important since if only a few prefetches are issued, their overall impact is low, even at high accuracy for these prefetches. We also compare the two history-based schemes, *QuickMine* and *C-Miner**, in more detail. Specifically, we show the benefits of context awareness and the benefit of incremental mining, versus static mining.

Detailed Comparison of Prefetching Effectiveness:

In Figure 8, we show the number of prefetches issued, and their corresponding accuracy for all prefetching algorithms. For TPC-W¹⁰ with a 512MB cache, shown in Figure 8(a), *QuickMine* issues 2M prefetches, while *C-Miner**, SEQ, and RUN issue less than 500K prefetches. The RUN scheme is the least accurate (< 50%) since many prefetches are spuriously triggered. The SEQ scheme exhibits a better prefetch accuracy of between 50% and 75% for the three cache sizes. Both *QuickMine*

and *C-Miner** achieve greater than 75% accuracy. While *C-Miner** has slightly higher accuracy than *QuickMine* for the rules issued, this accuracy corresponds to substantially fewer rules than *QuickMine*. This is because, many of the *C-Miner** correlation rules correspond to false correlations at the context switch boundary, hence are not triggered at runtime. As a positive side-effect, the higher number of issued prefetches in *QuickMine* allows the disk scheduler to re-order the requests for optimizing seeks, thus reducing the average prefetch latency. As a result, average promote latencies are significantly lower in *QuickMine* compared to *C-Miner**, specifically, 600 μ s versus 2400 μ s for the 512MB cache. For comparison, a cache hit takes 7 μ s on average and a cache miss takes 3200 μ s on average, for all algorithms.

For RUBiS¹⁰ with a 512MB cache, shown in Figure 8(c), *QuickMine* issues 1.5M prefetches, which is ten times more than *C-Miner** and SEQ. In the RUN scheme, the spatial locality of RUBiS causes more prefetches (250K) to be issued compared to SEQ, but only 38% of these are accurate, as shown in Figure 8(d). As before, while *C-Miner** is highly accurate (92%) for the prefetches it issues, substantially fewer correlation rules are matched at runtime compared to *QuickMine*, due to false correlations. With larger cache sizes, there is less opportunity for prefetching, because there are fewer storage cache misses, but at all cache sizes *QuickMine* issues more prefetch requests than other prefetching schemes. Similar as for TPC-W, the higher number of prefetches results in a lower promote latency for *QuickMine* compared to *C-Miner** i.e., 150 μ s versus 650 μ s for RUBiS in the 512MB cache configuration.

Benefit of Context Awareness: We compare the total number of correlation rules generated by frequent sequence mining, with and without context awareness. In our evaluation, we isolate the impact of context awareness from other algorithm artifacts, by running *C-Miner** without rule pruning, on its original access traces of RUBiS and DBT-2, and the *de-tangled* access traces of the same. In the *de-tangled* trace, the accesses are separated by thread identifier, then concatenated. We notice an order of magnitude reduction in the number of rules generated by *C-Miner**. Specifically, on the original traces, *C-Miner** without rule pruning generates 8M rules and 36M rules for RUBiS and DBT-2, respectively. Using the *de-tangled* trace, *C-Miner** without rule pruning generates 800K rules for RUBiS and 2.8M rules for DBT-2. These experiments show that context awareness reduces the number of rules generated, because it avoids generating false block correlation rules for the blocks at the context switch boundaries.

Another benefit of context-awareness is that it makes parameter settings in *QuickMine* insensitive to the concurrency degree. For example, the value of the looka-

head/gap parameter correlates with the concurrency degree in context oblivious approaches, such as *C-Miner**, i.e., needs to be higher for a higher concurrency degree. In contrast, *QuickMine*'s parameters, including the value of the lookahead parameter, are independent of the concurrency degree; they mainly control the *prefetch aggressiveness*. While the ideal *prefetch aggressiveness* does depend on the application and environment, *QuickMine* has built-in dynamic tuning mechanisms that make it robust to overly aggressive parameter settings.

The ability to dynamically tune the prefetching decisions at run-time is yet another benefit of context-awareness. For example, in TPC-W, *QuickMine* automatically detects that the *BestSeller* and the symbiotic pair of *Search* and *NewProducts* benefit the most from prefetching, while other queries in TPC-W do not. Similarly, it detects that only the *StockLevel* transaction in DBT-2 benefits from prefetching. Tracking the prefetching benefit per context allows *QuickMine* to selectively disable or throttle prefetching for low performing query templates thus avoiding unnecessary disk congestion caused by useless prefetches. In particular, this feature allows *QuickMine* to provide a small benefit for DBT-2, while *C-Miner** degrades the application performance.

Benefit of Incremental Mining: We show the benefit of dynamically and incrementally updating correlation rules through the use of the LRU based rule cache as in *QuickMine* versus statically mining the entire sequence database to generate association rules as in *C-Miner** [24, 25]. Figure 9 shows the number of promotes, cumulatively, over the duration of the experiment for *C-Miner**, *C-Miner** with periodic retraining (denoted as *C-Miner⁺*) and *QuickMine*. For these experiments, we train *C-Miner** on the *de-tangled* trace to eliminate the effects of interleaved I/O, hence make it comparable with *QuickMine*. As Figure 9 shows, the change in the access patterns limits the prefetching effectiveness of *C-Miner**, since many of its mined correlations become obsolete quickly. Thus, no new promotes are issued after the first 10 minutes of the experiment. In *C-Miner⁺*, where we retrain *C-Miner** at the 10 minute mark, and at the 20 minute mark of the experiment, the effectiveness of prefetching improves. However, *C-Miner⁺* still lags behind *QuickMine*, which adjusts its rules continuously, on-the-fly. By dynamically aging old correlation rules and incrementally learning new correlations, *QuickMine* maintains a steady performance throughout the experiment. The dynamic nature of *QuickMine* allows it to automatically and gracefully adapt to the changes in the I/O access pattern, hence eliminating the need for explicit re-training decisions.

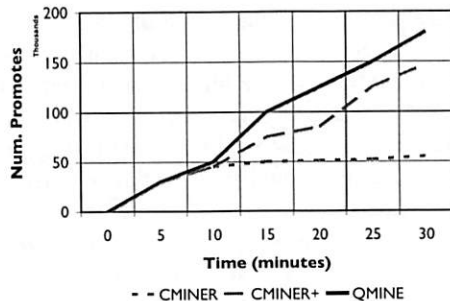


Figure 9: **Incremental Mining.** When access patterns change, the performance of static mining deteriorates over time.

7 Related Work

This section discusses related techniques for improving caching efficiency at the storage server, including: i) collaborative approaches like our own, which pass explicit hints between client and storage caches, or require more extensive code restructuring and reorganization, ii) gray-box approaches, which infer application patterns at the storage based on application semantics known *a priori*, and iii) black box approaches, which infer application patterns at the storage server in the absence of any semantic information.

Explicitly Collaborative Approaches. Several approaches pass explicit hints from the client cache to the storage cache [6, 12, 23, 28]. These hints can indicate, for example, the reason behind a write block request to storage [23], explicit demotions of blocks from the storage client to the server cache [41], or the relative importance of requested blocks [7]. These techniques modify the interface between the storage client and server, by requiring that an additional identifier (representing the hint) be passed to the storage server. Thus, similar to QuickMine, these techniques improve storage cache efficiency through explicit context information. However, as opposed to our work, inserting the appropriate hints needs thorough understanding of the application internals. For example, Li et al. [23] require the understanding of database system internals to distinguish the context surrounding each block I/O request. In contrast, we use readily available information within the application about preexisting contexts.

In general, collaboration between the application and storage server has been extensively studied in the context of database systems, e.g., by providing the DBMS with more knowledge of the underlying storage characteristics [32], by providing application semantic knowledge to storage servers i.e., by mapping database relations to objects [33], or by offloading some tasks to the storage server [31]. Other recent approaches in

this area [4, 11, 15] take advantage of context information available to the database query optimizer [11, 15], or add new middleware components for exploiting explicit query dependencies e.g., by SQL code re-writing to group related queries together [4]. Unlike our technique, these explicitly collaborative approaches require substantial restructuring of the database system, code reorganization in the application, or modifications to the software stack in order to effectively leverage semantic contexts. In contrast, we show that substantial performance advantage can be obtained with minimal changes to existing software components and interfaces.

Gray-box Approaches. Transparent techniques for storage cache optimization leverage I/O meta-data, or application semantics known *a priori*. Meta-data based approaches include using file-system meta-data, i.e., distinguishing between i-node and data blocks explicitly, or using filenames [5, 20, 22, 35, 43], or indirectly by probing at the storage client [2, 3, 37]. Alternative techniques based on application semantics leverage the *program backtrace* [12], user information [43], or specific characteristics, such as in-memory addresses of I/O requests [8, 18] to classify or infer application patterns.

Sivathanu et al. [36] use minimally intrusive instrumentation to the DBMS and log snooping to record a number of statistics, such that the storage system can provide cache exclusiveness and reliability for the database system running on top. However, this technique is DBMS-specific, the storage server needs to be aware of the characteristics of the particular database system.

Graph-based prefetching techniques based on discovering correlations among files in filesystems [13, 21] also fall into this category, although they are not scalable to the number of blocks typical in storage systems [24].

In contrast to the above approaches, our work is generally applicable to any type of storage client and application; any database and file-based application can benefit from QuickMine. We can use arbitrary contexts, not necessarily tied to the accesses of a particular user [43], known application code paths [12], or certain types of meta-data accesses, which may be client or application specific.

Black Box Approaches. Our work is also related to caching/prefetching techniques that treat the storage as a black box, and use fully transparent techniques to infer access patterns [10, 17, 26, 27]. These techniques use sophisticated sequence detection algorithms for detecting application I/O patterns, in spite of access interleaving due to concurrency at the storage server. In this paper, we have implemented and compared against two such techniques, *run-based prefetching* [17], and *C-Miner** [24, 25]. We have shown that the high concurrency degree common in e-commerce applications makes these techniques ineffective. We have also argued

that *QuickMine*'s incremental, dynamic approach is the most suitable in modern environments, where the number of applications, and number of clients for each application, hence the degree of concurrency at the storage server vary dynamically.

8 Conclusions and Future Work

The high concurrency degree in modern applications makes recognizing higher level application access patterns challenging at the storage level, because the storage server sees random interleavings of accesses from different application streams. We introduce *QuickMine*, a novel caching and prefetching approach that exploits the knowledge of logical application sequences to improve prefetching effectiveness for storage systems.

QuickMine is based on a minimally intrusive method for capturing high-level application contexts, such as an application thread, database transaction, or query. *QuickMine* leverages these contexts at the storage cache through a dynamic, incremental approach to I/O block prefetching.

We implement our context-aware, incremental mining technique at the storage cache in the Network Block Device (NBD), and we compare it with three state-of-the-art context-oblivious sequential and non-sequential prefetching algorithms. In our evaluation, we use three dynamic content applications accessing a MySQL database engine: the TPC-W e-commerce benchmark, the RUBiS auctions benchmark and DBT-2, a TPC-C-like benchmark. Our results show that context-awareness improves the effectiveness of block prefetching, which results in reduced cache miss rates by up to 60% and substantial reductions in storage access latencies by up to a factor of 2, for the read-intensive TPC-W and RUBiS. Due to the write intensive nature and rapidly changing access patterns in DBT-2, *QuickMine* has fewer opportunities for improvements in this benchmark. However, we show that our algorithm does not degrade performance by pruning useless prefetches for low performing contexts, hence avoiding unnecessary disk congestion, while gracefully adapting to the changing application pattern.

We expect that context-aware caching and prefetching techniques will be of most benefit in modern data center environments, where the client load and number of co-scheduled applications change continuously, and largely unpredictably. In these environments, a fully on-line, incremental technique, robust to changes, and insensitive to the concurrency degree, such as *QuickMine*, has clear advantages. We believe that our approach can match the needs of many state-of-the-art database and file-based applications. For example, various persistence solutions, such as *Berkeley DB* or Amazon's *Dynamo* [9], use a mapping scheme between logical identifiers and physi-

cal block numbers e.g., corresponding to the MD5 hash function [9]. Extending the applicability of our *QuickMine* algorithm to such logical to physical mappings is an area of future work.

Acknowledgements

We thank the anonymous reviewers for their comments, Livio Soares for helping with the Linux kernel changes, and Mohamed Sharaf for helping us improve the writing of this paper. We also acknowledge the generous support of the Natural Sciences and Engineering Research Council of Canada (NSERC), through an NSERC Canada CGS scholarship for Gokul Soundararajan, and several NSERC Discovery and NSERC CRD faculty grants, Ontario Centers of Excellence (OCE), IBM Center of Advanced Studies (IBM CAS), IBM Research, and Intel.

References

- [1] Transaction processing council. <http://www.tpc.org>.
- [2] ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Information and Control in Gray-Box Systems. In *Proc. of the 18th ACM Symposium on Operating System Principles* (October 2001), pp. 43–56.
- [3] BAIRAVASUNDARAM, L. N., SIVATHANU, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proc. of the 31st International Symposium on Computer Architecture* (June 2004), pp. 176–187.
- [4] BOWMAN, I. T., AND SALEM, K. Optimization of Query Streams Using Semantic Prefetching. *ACM Transactions on Database Systems* 30, 4 (December 2005), pp. 1056–1101.
- [5] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrated prefetching and caching strategies. In *Proc. of the International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS* (1995), pp. 188–197.
- [6] CHANG, F. W., AND GIBSON, G. A. Automatic I/O hint generation through speculative execution. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (1999), pp. 1–14.
- [7] CHEN, Z., ZHANG, Y., ZHOU, Y., SCOTT, H., AND SCHIEFER, B. Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (June 2005), pp. 145–156.
- [8] CHEN, Z., ZHOU, Y., AND LI, K. Eviction-based Cache Placement for Storage Caches. In *Proc. of the USENIX Annual Technical Conference, General Track* (June 2003), pp. 269–281.
- [9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM Symposium on Operating Systems Principles* (2007), pp. 205–220.
- [10] DING, X., JIANG, S., CHEN, F., DAVIS, K., AND ZHANG, X. Diskseen: Exploiting disk layout and access history to enhance I/O prefetch. In *USENIX Annual Technical Conference* (2007), pp. 261–274.

- [11] GAO, K., HARIZOPOULOS, S., PANDIS, I., SHKAPENYUK, V., AND AILAMAKI, A. Simultaneous pipelining in QPipe: Exploiting work sharing opportunities across queries. In *Proc. of the 22nd International Conference on Data Engineering, ICDE* (April 2006), pp. 162–174.
- [12] GNIADY, C., BUTT, A. R., AND HU, Y. C. Program-counter-based pattern classification in buffer caching. In *Proc. of the 6th Symposium on Operating System Design and Implementation* (2004), pp. 395–408.
- [13] GRIFFIOEN, J., AND APPLETON, R. Reducing file system latency using a predictive approach. In *Proc. of the USENIX Summer Technical Conference* (1994), pp. 197–207.
- [14] HAAS, L. M., KOSSMANN, D., AND URSU, I. Loading a cache with query results. In *Proc. of 25th International Conference on Very Large Data Bases* (1999), pp. 351–362.
- [15] HARIZOPOULOS, S., AND AILAMAKI, A. StagedDB: Designing database servers for modern hardware. *IEEE Data Eng. Bull.* 28, 2 (2005), 11–16.
- [16] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. *ACM Transaction on Database Systems* 26, 1 (2001), 96–143.
- [17] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems* 23, 4 (2005), 424–473.
- [18] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 14–24.
- [19] KEETON, K., ALVAREZ, G., RIEDEL, E., AND UYSAL, M. Characterizing I/O-intensive workload sequentiality on modern disk arrays. In *4th Workshop on Computer Architecture Evaluation using Commercial Workloads* (2001).
- [20] KROEGER, T. M., AND LONG, D. D. E. Design and implementation of a predictive file prefetching algorithm. In *Proc. of the USENIX Annual Technical Conference* (2001), pp. 105–118.
- [21] KUENNING, G. H., POPEK, G. J., AND REIHER, P. L. An analysis of trace data for predictive file caching in mobile computing. In *Proc. of the USENIX Summer Technical Conference* (1994), pp. 291–303.
- [22] LEI, H., AND DUCHAMP, D. An analytical approach to file prefetching. In *Proc. of the USENIX Annual Technical Conference* (1997), pp. 21–21.
- [23] LI, X., ABOULNAGA, A., SALEM, K., SACHEDINA, A., AND GAO, S. Second-Tier Cache Management Using Write Hints. In *Proc. of the Conference on File and Storage Technologies* (December 2005).
- [24] LI, Z., CHEN, Z., SRINIVASAN, S. M., AND ZHOU, Y. C-Miner: Mining Block Correlations in Storage Systems. In *Proc. of the FAST '04 Conference on File and Storage Technologies* (San Francisco, California, USA, March 2004), pp. 173–186.
- [25] LI, Z., CHEN, Z., AND ZHOU, Y. Mining Block Correlations to Improve Storage Performance. *ACM Transactions on Storage* 1, 2 (May 2005), 213–245.
- [26] LIANG, S., JIANG, S., AND ZHANG, X. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Proc. of the 27th IEEE International Conference on Distributed Computing Systems* (2007), p. 64.
- [27] MADHYASTHA, T. M., GIBSON, G. A., AND FALOUTSOS, C. Informed Prefetching of Collective Input/Output Requests. In *Proc. of the ACM/IEEE Conference on Supercomputing: High Performance Networking and Computing* (1999).
- [28] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. of the 15th ACM Symposium on Operating System Principles* (1995), pp. 79–95.
- [29] PHILLIPS, L., AND FITZPATRICK, B. Livejournal's backend and memcached: Past, present, and future. In *Proc. of the 19th Conference on Systems Administration* (2004).
- [30] RAAB, F. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark Handbook for Database and Transaction Systems* (2nd Edition). 1993.
- [31] RIEDEL, E., FALOUTSOS, C., GIBSON, G. A., AND NAGLE, D. Active disks for large-scale data processing. *IEEE Computer* 34, 6 (2001), 68–74.
- [32] SCHINDLER, J., GRIFFIN, J. L., LUMB, C. R., AND GANGER, G. R. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proc. of the Conference on File and Storage Technologies* (January 2002), pp. 259–274.
- [33] SCHLOSSER, S. W., AND IREN, S. Database Storage Management with Object-based Storage Devices. In *Workshop on Data Management on New Hardware* (June 2005).
- [34] SHEN, K., YANG, T., CHU, L., HOLLIDAY, J., KUSCHNER, D. A., AND ZHU, H. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems* (March 2001), pp. 197–208.
- [35] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-Safe Disks. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, USA, November 2006).
- [36] SIVATHANU, M., BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Database-Aware Semantically-Smart Storage. In *Proc. of the FAST '05 Conference on File and Storage Technologies* (December 2005), pp. 239–252.
- [37] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-Smart Disk Systems. In *Proc. of the FAST '03 Conference on File and Storage Technologies* (March 2003).
- [38] SRIKANT, R., AND AGRAWAL, R. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the 5th International Conference on Extending Database Technology* (March 1996), pp. 3–17.
- [39] WILKES, J. Traveling to Rome: QoS specifications for automated storage system management. In *Proc. of 9th International Workshop on Quality of Service* (2001), pp. 75–91.
- [40] WONG, M., ZHANG, J., THOMAS, C., OLMSTEAD, B., AND WHITE, C. *Database Test 2 Architecture*, 0.4 ed. Open Source Development Lab, http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/dbt_2_architecture.pdf, June 2002.
- [41] WONG, T. M., AND WILKES, J. My Cache or Yours? Making Storage More Exclusive. In *Proc. of the USENIX Annual Technical Conference, General Track* (June 2002), pp. 161–175.
- [42] YAN, X., HAN, J., AND AFSHAR, R. CloSpan: Mining closed sequential patterns in large databases. In *Proc. of the 3rd SIAM International Conference on Data Mining* (2003).
- [43] YEH, T., LONG, D. D. E., AND BRANDT, S. A. Increasing predictive accuracy by prefetching multiple program and user specific files. In *Proc. of the 16th Annual International Symposium on High Performance Computing Systems and Applications* (2002), pp. 12–19.

Free Factories: Unified Infrastructure for Data Intensive Web Services

Alexander Wait Zaranek, Tom Clegg, Ward Vandewege, George M. Church
Harvard University
await@genetics.med.harvard.edu

Abstract

We introduce the Free Factory, a platform for deploying data-intensive web services using small clusters of commodity hardware and free software. Independently administered virtual machines called Freegols give application developers the flexibility of a general purpose web server, along with access to distributed batch processing, cache and storage services. Each cluster exploits idle RAM and disk space for cache, and reserves disks in each node for high bandwidth storage. The batch processing service uses a variation of the MapReduce model. Virtualization allows every CPU in the cluster to participate in batch jobs. Each 48-node cluster can achieve 4-8 gigabytes per second of disk I/O. Our intent is to use multiple clusters to process hundreds of simultaneous requests on multi-hundred terabyte data sets. Currently, our applications achieve 1 gigabyte per second of I/O with 123 disks by scheduling batch jobs on two clusters, one of which is located in a remote data center.

1 Introduction

We built Free Factories to help the PGx team win the Archon X PRIZE for Genomics and to meet the needs of the Personal Genome Project. The prize is awarded for sequencing one hundred complete human genomes in less than ten days [29]. Doing this with Polony sequencing [17, 25] and related technologies [30, 31], as the PGx team plans to do, will involve distilling many petabytes of raw data to produce about 100 gigabytes of output. This DNA sequencing capacity can be used to help build a database of personal genome-phenome data sets; coupled with a data mining and analysis engine, this will provide opportunities for many new discoveries.

Storing and analyzing data at this scale still requires exotic computing systems [3, 15, 28] – many scientists, physicians and members of the general public would like to participate in the development of these technologies,

but do not have access to the resources they need to get started. Furthermore, anyone creating a database of sensitive personal information has to address privacy and disclosure concerns, and there is no single correct way to do that. The Personal Genome Project aims to overcome these obstacles and, ultimately, give individuals the tools to make genetic discoveries of their own [7, 19, 20].

To help alleviate these barriers, we consider the needs of a relatively small organization that supports several independent applications. Some such organizations support scientists in research activities, like developing new sequencing technologies; some focus on medical applications, like predicting which treatments will be effective for a particular patient. Within each of these communities, scientists and application developers generally benefit by sharing data and computing resources, although they may need to segregate some data and resources in

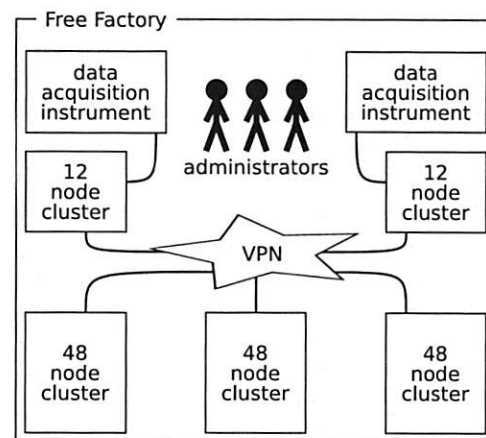


Figure 1: A Free Factory contains about five clusters of 12 to 48 nodes. Some clusters are colocated with data acquisition instruments; their sizes are limited by the available power, cooling, and space. Other clusters are located in data centers. The clusters are interconnected by relatively slow networks.

order to satisfy a particular security model. End users choose who to trust with their personal data, establish legal agreements with those trustees to control how their data is used, and access relevant information using web services.

With the Free Factories platform, we make the first steps toward this vision. We emphasize efficiency in a range of installation sizes, from a single 12-node cluster to several 48-node clusters. We cater to data-intensive applications that are conducive to parallel computation, but are limited by the ability of storage systems to support many concurrent tasks. We avoid proprietary software and expensive hardware. Organizations can start working with substantial data sets on small clusters, and expand their capacity by adding new clusters. Multiple small clusters also provide opportunities to enforce different privacy and data integrity guarantees for different applications and data.

Small installations of low-cost hardware provide processing and storage capacity at the scale we need for these applications, but efficient and fault-tolerant utilization of this hardware is non-trivial. We have used a pragmatic approach of selecting and arranging free software to make the best of the performance characteristics of cheap hardware. Our goal is to utilize 90% of our hardware capacity, including disk I/O bandwidth, network bandwidth, and CPU time. The result is a unified platform that makes efficient use of hardware in an environment where a variety of users and applications share storage and computation resources. We encourage others to deploy and develop this platform further. [16]

2 Design and architecture

A Free Factory provides hosting, data storage, and batch processing services for a number of web applications. These applications involve data-intensive computation: they are conducive to asynchronous parallel processing, but their performance is limited by the available disk I/O bandwidth. Their demands for CPU time are highly variable, so it is sensible for them to share a pool of CPU resources by submitting batch jobs. They also tend to share data sets with one another, so it is sensible to share a large data storage system. The application developers have common goals rather than being in competition, so it is beneficial to let them see the source code and results of one another's batch processing jobs. The applications themselves may be maintained by different development teams, so each application should run in its own independent virtual machine.

We identify the following roles in the Free Factory environment. "Users" – scientists, physicians, and members of the general public – are interested in a web service and interact with it via a web browser. "Administrators"

maintain the Free Factory infrastructure. A "trustee" sets policy and obtains funds to pay for staff, hardware, and hosting. "Developers" are the application developers and scientists who maintain Freegols. "Freegols" are web services that utilize cluster computing and storage resources. The term Freegol, or Free Golem, emphasizes the idea that the web services are developed and maintained independently of the cluster infrastructure, and independently of one another.

The canonical Free Factory (Figure 1) contains about five clusters with 12 to 48 nodes each. Some clusters can be co-located with data acquisition engines such as DNA sequencing instruments. Each cluster acts as a web hosting platform for several applications, as well as supporting the data storage and batch processing needs of those applications. A Free Factory of this size can be maintained by three administrators.

Each cluster is constructed using 1U rack-mount machines with big disks and inexpensive CPUs. Today, each of these low-cost machines offers about 240 MB/s of disk I/O bandwidth as well as 2 Gb/s of network bandwidth. With data and processing resources striped across an entire 48-node cluster with 192 disks, it is theoretically possible to achieve 11 GB/s of disk I/O during a batch job. The two clusters we have built contain 85 and 38 disks respectively. At 60 MB/s per disk this gives us 5.1 GB/s and 2.2 GB/s of available disk bandwidth respectively.

We use virtualization to deploy cache, storage, and batch processing services on every single node in each cluster: CPU-intensive jobs can make use of every CPU in the cluster, while data-intensive jobs can make use of every disk. This layout allows us to achieve high I/O throughput even while many concurrent processes are working on the same data set. We have achieved as much as 1 GB/s of I/O on a cluster with many concurrent processes; this compares favorably with a 12-disk RAID-6 system, which we have to limit to a single reader in order to achieve a sustained throughput of 100 MB/s.

On each machine, a "warehouse instance" runs the processes that implement cache, storage, and batch processing services. Warehouse instances are implemented as virtual machines on the nodes that are used for hosting Freegols, and consume entire physical machines in other cases. Each cluster manages its own cache, storage, and batch processing services using a number of controller processes that run in a virtual machine. Freegols and batch programs use the warehouse client library to communicate with these controller processes and the warehouse instances' service processes (Figure 2).

2.1 Commodity hardware, free software

When building an affordable, high-availability, data-intensive web service it is desirable to have control of the

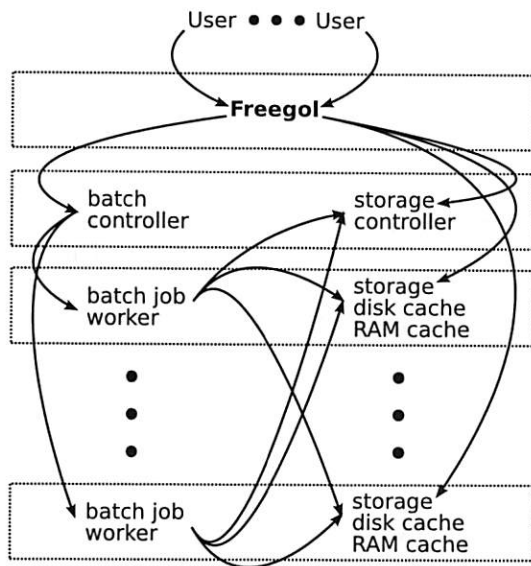


Figure 2: Dotted lines denote virtual machines. Batch processing workers in warehouse instances are dispatched by the batch controller on behalf of Freegols. RAM cache, disk cache, and long term storage services are accessed by Freegols and batch jobs using the same client library.

system's total cost of ownership. Part of our strategy is to avoid proprietary technologies in favor of free software. This way, we can build on existing tools and have confidence that we can replace or modify them when necessary.

When choosing hardware, we are interested in maximizing the usable disk, RAM, CPU, and network bandwidth per unit cost. At present dual gigabit ethernet, one terabyte SATA disks, and dual-socket quad core motherboards seem to best fit our needs. (We prefer larger disks even at a higher cost per gigabyte because disks have high failure rates independent of size [2, 12, 23, 24] and manual intervention is expensive even in a fault-tolerant system.) Full-bandwidth 48-port switches are also available at low cost. Therefore, the most affordable way to configure a large number of disks and CPUs today is to build a number of 48-node clusters, interconnected by relatively slow network links or virtual private networks.

Given the limited size of each cluster, scalability requires that applications have access to more than just one cluster. We expose the network topology to the applications so that they can make informed decisions about where and when to perform computation, and where to store data, depending on the varying availability of these resources on different clusters. (Our intent is for future client libraries to help applications make good scheduling decisions; currently it is practical for a developer to select one of our two clusters when running a job.)

Each cluster is self-contained; hardware and network failures do not cascade to other clusters. The small cluster size makes it feasible to deploy entire clusters at once, rather than performing incremental upgrades to a large cluster. Multiple clusters can be used to increase confidence in the repeatability of computational results, and to monitor the effects of different combinations of hardware, software, and usage patterns.

To illustrate the cost of a Free Factory we consider purchasing a 48-node cluster with 192 1 TB disks for \$170,000. Annual operating costs include \$27,000 of power (18 kW at \$0.17/kWh), \$25,000 for network access and floor space (at the rate we pay at Harvard), and \$50,000 for a part-time administrator. Thus, the total cost to deploy the cluster is \$272,000 for the first year, and \$102,000 per year thereafter.

For a point of reference, we consider Amazon Web Services, a popular computing platform that allows an organization to pay for computation and storage on an as-needed basis. This is often less expensive than using dedicated hardware because the cost of processing is determined by average demand rather than peak demand. However, for the data-intensive applications we consider here, the strategy of frequently allocating and releasing compute nodes is less beneficial because of the time spent copying data to and from the nodes each time. In effect, a CPU-on-demand system requires a higher allocation rate in order to do the same work, compared to a dedicated hardware approach where data is kept close to the processors and can be read at full speed whenever it is needed.

We overlook this distinction for the sake of making a direct comparison with the Amazon EC2 and S3 services [1]. Amazon EC2 offers an "extra large instance" with two 1 TB disks and four virtual CPUs for \$0.80 per hour. Thus, a 48-node cluster is roughly equivalent to 96 extra large instances. If the cluster achieves 25% CPU utilization, its value is comparable to 24 extra large instances at \$168,000 per year. Meanwhile, S3 provides long term storage for \$0.15/GB. At this rate, it costs an additional \$43,000 per year to store 24 TB of data (half of the long term storage capacity of the cluster). The actual amount of data transferred to and from S3 depends on the application; if 15 TB is transferred to Amazon at \$0.10/GB, and 2 TB is transferred out of Amazon at \$0.17/GB, then the transfer cost is \$1,840 (traffic between EC2 and S3 is free). The total cost of the Amazon service over two years is \$424,000, compared to \$374,000 for the first two years of a Free Factory.

2.2 Freegols and virtualization

There are a wide variety of languages, toolkits and methodologies for deploying scalable web services. One

factor that contributes to Amazon EC2's popularity is that it permits web service developers to choose their own tools. We found that giving developers this freedom suited our environment too.

Virtualization encourages a model wherein developers have "root" privileges on their own virtual servers, or Freegols. Typically, a Freegol is configured as an Ubuntu server with common application server software like Apache and MySQL. The warehouse client library is easy to install and upgrade using the native package manager. Our goal is to make it easy for developers to start using Freegols to deploy services; part of this strategy will be to port the Perl client library to other popular languages like Python.

RAM, virtual processors, and network bandwidth are shared among Freegols, cache and storage service processes, and batch jobs. If necessary, a developer can ensure that a Freegol does not share these resources with other Freegols by getting an allocation for all of the available CPU and RAM.

In addition to Freegols, it is often beneficial to set up virtual machines on the cluster for applications that do not use the cache, storage, or processing services. Cluster administrators are likely to use virtual machines to deploy common network services like web proxies, DNS caches, and backup servers.

2.3 Cache and storage services

The objectives of our storage services are to: (1) minimize I/O bottlenecks in order to make the best use of available CPU cycles; (2) provide a low-latency shared cache with automatic garbage collection; (3) provide long term storage with high read and write throughput and provisions for usage accounting. The storage services must yield good performance when used directly from Freegols, as well as from batch jobs. Inspired by the Google File System [13], Bigtable [6] and the plethora of raw materials made possible by free and open source software, we felt we could build a system that suited our needs perfectly.

We implement a three-level content addressable storage service. We use Memcached [9] as a low latency RAM cache. This is well suited to small strings (less than one megabyte) and it works well even with many concurrent clients because it does not employ a central controller. We use MogileFS [9] to implement a cluster-wide distributed disk cache. This gives good performance for block sizes up to 64 MiB. For long term storage we have developed software that minimizes the role of a central controller while providing opportunities for usage accounting. All of these storage and cache services are accessible to all applications and batch jobs in the Free Factory.

Aggregate I/O bandwidth is limited by several factors. Our storage services are designed to minimize the effects of these factors.

1. Disk seeks reduce aggregate disk read and write bandwidth. We minimize seeks by storing data in contiguous 64 MiB segments when possible, and ensuring that each segment read/write operation is not interrupted by any other disk activity. This means that readers tend to wait longer before they start to receive data, which is why general purpose operating systems do not use this strategy; however, in this environment, high throughput is more valuable than low latency.
2. A gigabit network interface can only handle two concurrent readers at full disk transfer rates. To prevent this from limiting throughput when many concurrent processes are accessing the same data set, we stripe every data set across all of the cluster nodes.
3. Central controllers get bogged down when they try to handle too many concurrent clients. Our storage service can read and write blocks without involving the storage controller in real time. Our disk cache is indexed in RAM, so most reads do not involve the controller.
4. Writes are slow because robustness requires storing multiple copies of each block. When an identical copy of an output block already exists in the storage service, our content-addressing approach allows the client library to transparently skip unnecessary write operations.

The cache and storage services use an MD5 addressing approach: the name of each block of data is the MD5 checksum of the data. This naming scheme provides several benefits.

1. The client library, when retrieving a block from the cache and storage services, computes the MD5 checksum of the data and compares it with the block name. If the checksum does not match, the client library can try to fetch the data from a different host, a different storage service, or a remote cluster. This verification process is completely transparent to the application.
2. Multiple jobs often produce the same output. For example, a Freegol may re-run a job every time the job's source code is modified in the source repository, and every time an operating system upgrade is performed, to make sure the job still produces the same result. If the output is identical to the previous run, no additional storage space is consumed

by the new job. We encourage developers to make use of this property by segregating their main output from their diagnostic messages (which are likely to contain timestamps and the like, but are usually small) and by avoiding non-deterministic outputs (sometimes this involves simple tricks like using the “Minimal” option in the `IO::Compress::Gzip` Perl module).

3. Freegols and concurrent batch jobs can share data without worrying about overwriting blocks at inopportune times.

Our simple tests have demonstrated that reading 64 MiB files sequentially results in throughput exceeding 90% of a disk’s maximum sustained transfer rate. Real data used by Freegols, on the other hand, is likely to include many smaller files. To help Freegols achieve high throughput when working with smaller files, we introduce a “manifest” file format. In addition to increasing performance, the manifest format is a valuable tool for managing large data sets.

A manifest is an index to a collection of data files, analogous to a directory tree in a traditional filesystem. It is stored as a plain text file. Each line of the text file represents a “stream”; each stream contains a set of data files. The content of the data files is stored in a manner similar to a UNIX `tar` archive: the data from all files in a stream are concatenated, the result is split into 64 MiB blocks, and the blocks are written to cache or storage. The manifest file specifies the MD5 checksums and sizes of these data blocks, as well as the names of the individual files and their positions within the stream. The manifest file itself can be split into 64 MiB blocks and stored, and its unique key – the list of MD5 checksums of those blocks – can be used to retrieve it. (If this list of checksums is inconveniently long, the list itself can be stored in a separate block, whose MD5 checksum then serves as a more concise key to the large manifest.)

This manifest format has several noteworthy features. It is concise: a short key is enough to specify a large collection of data. It is portable: if two jobs running on different clusters produce identical output, the resulting manifest keys are also identical. The integrity of the data blocks, and the manifest itself, are easily verified. It is efficient to read an entire stream worth of data from disk, even if the stream represents many small files. However, random access – reading and writing small files in various streams out of order – is not efficient. We expect applications to be cognizant of this restriction, and read and write entire streams whenever possible.

Once a manifest is written to the cache or storage service, it can be retrieved, or used as the input to a batch job, by any Freegol that knows its key. Also, each cluster has a central database of manifest names. To attach a

name to a manifest, a Freegol sends a request to the cluster’s storage controller specifying the manifest key, the desired name, and – to avoid race conditions – the manifest key that was previously associated with the name. Naming a manifest has several consequences.

1. Any Freegol can look up the name to retrieve the manifest key.
2. The data set is considered to be valuable to the signer. If the blocks referenced by the manifest are in long term storage, those blocks should not be deleted.
3. The old manifest, if it is not associated with any other names, is no longer considered valuable; the data blocks mentioned in it may be deleted if they are not mentioned in any other named manifests.

Optionally, a Freegol may also specify a list of PGP keys indicating entities that have permission to overwrite this manifest name.

One drawback to content-addressable storage is that in-place updates are inefficient. For example, if a 32 MiB stream is written, and a new version of the stream is written afterward that has 16 MiB of additional data appended to the original 32 MiB, then both the 32 MiB version and the 48 MiB version may be written to disk. We find this acceptable for the following reasons. The manifest format allows the 48 MiB stream to be stored by referencing the original 32 MiB block followed by the new 16 MiB block, if the existence of the 32 MiB block is known when the second version is written. In any case, we are willing to sacrifice some storage space in order to avoid race conditions.

This illustrates one of the useful aspects of the manifest format. A manifest key specifies the data itself, not just a set of filenames. If a modified version of a large data set appears, the previous key can still be used to access the old data, and a program requesting the original version will never unexpectedly receive the newer data. This simplifies application design, and provides a significant practical benefit for scientific applications and other environments where repeatability is of major concern.

Periodic garbage collection is inexpensive. The storage controller can quickly read all of the manifests that appear in the manifest name database, and produce a list of blocks that are still in use. The 64 MiB block size ensures that the resulting list is small compared to the amount of stored data. Garbage collection is accomplished by comparing this list against the list of blocks stored on disk. We have not yet implemented automatic garbage collection but we have found that a list of 588,000 distinct block names, representing 11 TB of data referenced by 1151 distinct manifests, can be generated in 70 seconds.

2.4 Batch processing services

The objectives of our batch processing services are to: (1) use as many as possible of the available CPU cycles on all machines; (2) make it easy to repeat jobs many times on various clusters to check for bugs and inconsistencies; (3) handle occasional failures gracefully; (4) keep statistics about performance and failure rates.

Batch processing is coordinated by a batch controller on each cluster. The batch controller accepts requests from Freegols to schedule new jobs. The batch controller starts new jobs when the requested number of warehouse instances become available. Freegols can expect the batch controller to occasionally pause and resume a job, or reduce its resource allocation, depending on subsequent job submissions. (Our current implementation uses a simple greedy scheduling algorithm, and the batch controller only pauses and resumes jobs when specifically requested by a Freegol.)

Freegols can retrieve a list of current, pending, and previous jobs from the batch controller. This list includes specifications and statistics for each job, including inputs, outputs, start and finish times, and (for active jobs) what portion of the job has been completed so far. Freegols can poll the batch controller to determine the status of their own jobs, get hints about how busy the cluster is, and look up details of jobs that other Freegols have submitted.

The execution of a batch job is supervised by a job manager process running on the same virtual machine as the batch controller. The job manager supports a computation strategy similar to MapReduce [11]. Each job consists of a number of steps, each of which is performed on a single warehouse instance. Each job step stores some output in the cache; the job manager assembles the output into a manifest at the end of the job. Additionally, each job step has the ability to enqueue more job steps.

The program that performs the work of a single job step is called a “mr-function” (from “MapReduce function”). Mr-functions are kept in a revision control system. Administrators and developers can update existing mr-functions and create new ones, subject to access controls on the revision control repository. Once it is committed to the repository, a mr-function can be used in a job submission by any Freegol.

The most convenient way to construct a batch job is to use a single manifest as input, and schedule one job step for each stream in the manifest. Each job step reads one full stream from the input manifest from start to finish, and writes one full stream in the output manifest. The client library comes with tools and examples to make it easy to write mr-functions that use this strategy.

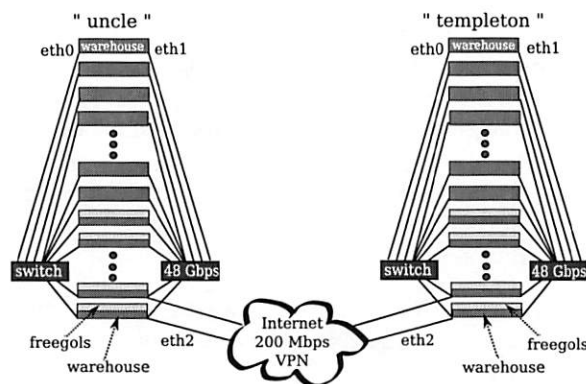


Figure 3: On each cluster, a few physical machines run the Xen hypervisor (mixed dark/light gray blocks). These machines are partitioned into warehouse instances, Freegols, and other virtual machines. Other physical machines are dedicated to warehouse instances (dark gray only).

3 Implementation

3.1 Commodity hardware, free software

We are currently operating two clusters using a variety of commodity off-the-shelf hardware, free software such as GNU/Linux [14] and our own custom software that is released under the GNU GPL [16]. Our two clusters are located a few kilometers apart and connected by Harvard University’s fiber optic network. “Uncle” is our experimental research cluster; “templeton” is our production cluster. Our clusters are depicted in Figure 3. Uncle is largely made up of 32-bit dual-CPU Intel Xeon machines, many of which are four years old. Templeton’s hardware is more recent: each machine has two dual-core AMD Opteron 64-bit processors.

Each cluster consists of 47 machines. All newer machines have four disk slots and the older machines are diskless. Each newer machine has two gigabit ethernet ports, which are connected to two 48-port gigabit switches. Two to four “headnodes” have a third ethernet port connected to the upstream switch, and optionally a fourth port connected to an out of band management network. The headnodes act as gateways to the internet and as VPN endpoints.

Our VPN is a simple OpenVPN [22] point-to-point setup, terminated on a headnode at each end. The VPN data rate reaches about 200 Mb/s. Throughput is limited by the processing speed of one endpoint that has one dual-core Opteron 265 CPU at 1.8 GHz. The other endpoint has 2 single-core Opteron 250 processors at 2.4 GHz; CPU load is considerably lower there.

Uncle currently runs the latest release of Ubuntu [5], a popular flavor of the Debian [27] GNU/Linux system that many lab members run on their desktops, while

Templeton runs the latest “long term support” release of Ubuntu. We chose Ubuntu over Debian because of Ubuntu’s predictable six month release schedule, but we expect to build a Debian cluster in the future. We like the Debian and Ubuntu philosophy and have found that packaging our software using Debian/Ubuntu tools to be a good way to automate the installation of the client library and its dependencies. We believe that inclusion in major community projects, such as Debian, is an excellent way to both reach a wider audience and to further improve our installation automation. Ultimately, we aim to be distribution agnostic.

The latest machines that we purchased came pre-installed with coreboot [8], a free software BIOS that has a number of advantages over proprietary alternatives. All source code is open and available under the GPL. Serial console support is reliable. Boot time is much faster: coreboot takes only a few seconds to bring the machine into a state where it can start booting the operating system. Also, we can exactly configure the platform to our needs, which allows us to make the platform both more reliable and more secure.

We use Opengear [21] CM4148 console servers on each cluster for out-of-band access to the serial console of each physical machine. The Opengear console servers are embedded Linux machines for which the entire source code is available for download. The company provides instructions for modifying the firmware, and for building the firmware from source. We also use networked power distribution units to allow remote power cycling of any device in the cluster via our out-of-band management network.

Aside from the Linux kernel, our software relies on many other open source packages. Notably, Slurm and Munge [18] provide authenticated inter-process communication between the warehouse instances and the batch controller. We rely on Slurm to track which warehouse instances are available for running jobs, and to allow the batch controller to execute batch job steps on the warehouse instances. It does this well, with low latency. It is also a convenient tool for administrative tasks like installing packages and updating configuration files on many instances at a time.

Memcached and MogileFS [9] provide the RAM and disk cache services respectively. MogileFS provides distributed storage with low-latency replication. Used in conjunction with Memcached, it performs well as long as there are not too many concurrent writes.

Perl modules from CPAN are used by the client library, controllers, and service programs for HTTP request handling, data compression, and MD5 hashing. The batch controller uses a MySQL database as a job queue and an archive of past jobs. Subversion provides revision control for the programs that run in the batch processing sys-

tem, as well as the client library and the service software itself.

3.2 Freegols and virtualization

Some physical machines are configured as warehouse instances, dedicated to providing processing, cache, and storage services. Others are partitioned into virtual machines using the Xen hypervisor – always with one virtual machine configured as a warehouse instance, along with one or more Freegols and other virtual machines controlled by the cluster administrators.

To keep configurations simple, we set aside 4 GiB of RAM on each warehouse instance for use in batch jobs, and allocate the remainder to Memcached processes.

On virtualized machines and dedicated warehouse instances alike, we use RAID-1 to protect all of the local filesystems. We have found that dedicating two entire disks to RAID-1 results in an excess of RAID-protected space. It is wasteful to allocate that space to the storage services, which can already accommodate disk and node failures without RAID-1. Linux allows us to partition the first two disks, assign one partition on each to a software RAID-1 array managed by the Linux Volume Manager, and allocate the remaining space to the MogileFS cache. This is more efficient than whole-disk mirrors, but it still forces us to commit to a partitioning scheme early on. Different permutations of Linux RAID and volume management tools could give us greater flexibility, but we have chosen to avoid the extra complexity that would result. In the future we hope iSCSI will provide more flexible options without creating too much work for administrators.

Developers, and some of their users, have access to shell accounts on their Freegols. Our security model is largely perimeter-based at this point, because our users are relatively trustworthy. Specifically, a cluster has one virtual machine with an unprivileged account that is shared by all users. To connect to the SSH port on a Freegol, a user must log in to this shared account using an SSH private key, and specify the name of the Freegol in a remote command string. The `authorized_keys` file in the shared account instructs the SSH server to ignore the client-supplied command and instead run a script that establishes a tunnel to the requested Freegol (the SSH server makes the supplied Freegol name available in an environment variable). Before establishing the tunnel, the script checks a list of permitted combinations of SSH public keys and Freegols. This login procedure is easy to express in a `ProxyCommand` directive in the user’s SSH client configuration file.

Virtual machines are used for deploying DNS caches and servers, an SMTP server for routing incoming mail, and a local Ubuntu mirror site. One virtual machine runs

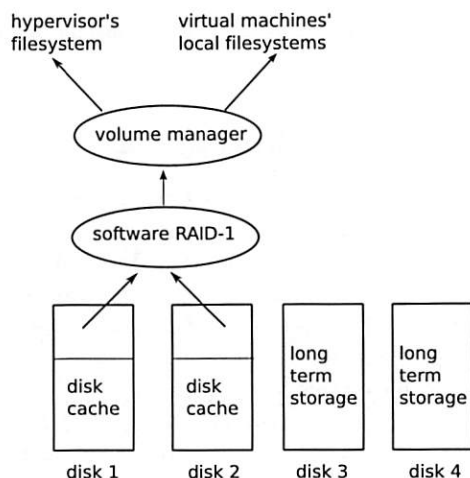


Figure 4: On a physical machine that is running several virtual machines, a software RAID-1 volume is exported to the Linux logical volume manager, which provides space for local filesystems. The remaining portions of the first two disks are used by the disk cache service. The third and fourth disks are dedicated to the cluster's long term storage service.

a dedicated reverse HTTP proxy. All HTTP and HTTPS traffic to the cluster is forwarded by the headnode to this proxy, which forwards each request to the appropriate Freegol and returns responses to the clients.

Since Harvard routinely receives high volume web traffic, we tested whether our setup can also support a high volume web service. We configured the virtual machine running our reverse proxy server with moderate resources: four shared Opteron 265 cores and 1 GiB RAM. We then launched 2,000,000 requests at the proxy server (with keepalive disabled to simulate a worst-case scenario), with 500 simultaneous requests each coming from two physical machines on our LAN. All requests were handled without errors and completed in 657 and 660 seconds on client 1 and 2 respectively. The mean time to complete a request was 0.3 seconds. Both clients completed 90% of requests within 0.2 seconds, and 99% of requests in 4.5 seconds. The slowest request from client 1 was completed in 93 seconds while the slowest request from client 2 was completed in 107 seconds. During this test, a few of us continued to use other Freegols from multiple points on the Internet and found no perceptible difference in response times. We conclude from this that a Freegol on our cluster can withstand popularity spikes without adversely affecting other Freegols.

The cache, storage, and processing services involve several databases and controller processes, which we run on a single virtual machine. It would also be possible to distribute these processes across several virtual machines; we have not thoroughly explored the perfor-

mance implications of this choice.

Developers frequently benefit from having separate “development” and “production” Freegols for a given application. The virtualization approach makes it easy for us to deploy these quickly at minimal cost.

3.3 Cache and storage services

The cache and storage services consist of three software layers (RAM cache, disk cache, and long term storage), a manifest name server, and a client library that is used by Freegols and batch jobs.

The client library contains most of the intelligence. It helps applications split and combine data into 64 MiB blocks; it chooses RAM or disk cache for different block sizes according to tunable settings; it constructs hashes when storing data; it avoids writing blocks to cache if they are already present; it stores each block on multiple warehouse instances when writing; and it constructs and parses manifests.

The client library is built on the assumption that all of the underlying storage services are unreliable. It verifies data integrity during retrieval operations. It attempts to retrieve blocks from alternate sources when data is corrupted or missing.

The aim of the storage services is to maintain the highest possible aggregate throughput when reading and writing blocks.

Our RAM cache uses Memcached. The Memcached client library and servers implement a distributed shared-nothing hash table. Requests for a name are mapped to weighted buckets by the Memcached client library, and then fulfilled by the Memcached server. We use the default library, which permits rehashing of blocks in the event that a server becomes temporarily unavailable, but requires that the entire cache be flushed if bucket sizes or the number of nodes change. In our system, such cache flushes should not be common because we deploy clusters with a fixed number of nodes by design. Memcached has a built-in limit of 1 MB for each data block, and our client library uses this as the default maximum size for cached items; however, if the application specifies a larger limit, the client library transparently splits larger blocks in 1 MB chunks when storing, and reassembles them when retrieving them.

The disk cache service is implemented using MogileFS. MogileFS uses a MySQL database to store the locations where each file is stored. This single database quickly becomes a performance bottleneck when it is accessed for every cache read and write operation on the entire cluster. Our client library alleviates this problem by using Memcached to cache the file locations: as a result, most read operations do not involve querying the MySQL database.

For long term storage, we implement a simple service called Keep. Each warehouse instance with long term storage space runs a network server that accepts HTTP “GET” and “PUT” requests. The client library is responsible for replication, fault tolerance, and load balancing as described below.

A “PUT” request is a signed request to copy data from cache to long term storage. The Keep server fetches the data itself from the local cache or from a remote cluster, according to the hints that come with the request. This approach is very efficient in the case where applications first store a lot of data in the disk cache, and later choose to keep some of that output in long term storage. This case has turned out to be so common that we have not yet implemented an operation that writes directly to long term storage.

When a “PUT” request results in a disk write, an accounting entry is also recorded, with the requestor’s IP address and cryptographic signature. Currently, the server does not verify the signature because this has not been necessary in our environment, but it does demand that each request arrive in the form of a PGP signed message. It also records a timestamp and the full text of the request; and it requires that the message include a timestamp within 5 minutes of the server’s system clock. We do not expect the overhead of checking signatures to cause an inordinate performance burden because of our large block size.

The client library uses MD5 checksums of data blocks to distribute them evenly among the available Keep servers. First, we specify that a given cluster has a fixed number of servers (there should be one on each cluster node). For a given block of data, we define eight preferred storage positions, derived from eight substrings of the block’s 128-bit MD5 checksum: portions of the checksum are used to compute a list of eight different Keep servers, and these Keep servers are tried one by one until enough copies have been written.

When a block has been written to Keep, the client library notes which of the eight preferred Keep nodes were used, and encodes this information as a hexadecimal number, where the least significant bit corresponds to the first preferred storage position. The block location is given as the letter “K” (from “Keep”) followed by the hexadecimal number, the symbol “@”, and the name of the cluster. For example, if a block is stored on the templeton cluster using the second and third servers in the probe order, the list of locations is encoded as “K06@templeton”. This string, along with the size of the block (an unusually short 3 byte block in this case) are appended to the MD5 checksum using the delimiter “+”:

```
acbd18db4cc2f85cedef654fccc4a4d8+3+K06@templeton
```

This resulting block name, stored in a manifest, provides enough information for a Freegol or a batch job to retrieve the block, regardless of which cluster it is stored on.

This notation provides an opportunity to support other storage systems – for example, “S” might designate blocks stored using Amazon’s S3 service – and to cite multiple clusters and storage systems, when a block is stored in multiple locations.

A “GET” request is much like a request for a static HTML page: the MD5 hash provided in the client’s request is the name of the disk file where the data is stored on the server. A Keep server may have up to four disks available for long term storage; in that case, it may have to perform four directory lookups in order to locate a file.

The storage controller maintains a database of manifest names and keys in a MySQL database. Any Freegol can connect to the storage controller’s “warehoused” network server program and retrieve the key currently associated with a given name, or the entire list of names and keys. A Freegol can also submit a signed request to update the database by changing the key for a given name, or adding a new name. Currently, the “warehoused” program does not verify signatures of these requests because this has not been necessary in our environment, but it does demand that each update request arrive in the form of a PGP signed message, and that the Freegol correctly specify the key currently associated with the name.

The client library includes functions for reading and writing blocks to the cache and storage services. The library also provides convenient functions for constructing manifests while storing data, reading streams and individual files from an existing manifest, looking up manifest keys by name, and updating the name database. Command line tools are provided for submitting jobs, looking up details of current and completed jobs, reading and writing data sets, and copying data sets from one cluster to another. These programs also serve as examples of how to use the client library.

The warehouse client library makes it convenient for Freegols and batch jobs to read and write data on remote clusters as well as the default local cluster. We provide this flexibility – rather than requiring data to be explicitly copied from cluster to cluster using a separate mechanism – for several reasons. It allows applications to achieve various levels of data redundancy, either synchronously or asynchronously, according to their needs. It supports the convenience of running jobs on remote clusters without arranging for all of the required data to be copied ahead of time. Generally, it follows the trend of doing a reasonable thing by default while offering the flexibility to accommodate the diverse needs of a variety of applications.

3.4 Batch processing services

The “warehoused” program accepts signed job submissions from Freegols, and answers queries about previously submitted jobs. The “mapinit” program starts queued jobs when instances become available, using Slurm’s `salloc` command to reserve warehouse instances. At the start of each job, the “mrjobmanager” program first retrieves the specified version of the appropriate “mr-function” from a Subversion repository, then invokes it on one of the allocated warehouse instances. The mr-function examines the input object (normally a manifest) and instructs mrjobmanager to queue a number of job steps.

During the course of a job, mrjobmanager allocates individual job steps to warehouse instances, monitors their output and exit codes to detect failures, and re-queues them when they fail. Each job step is expected to store some output blocks and send the blocks’ names to mrjobmanager by printing them to its standard error file descriptor. When all job steps have completed, mrjobmanager reads these blocks and assembles them into a final output stream. This final output stream is expected to be a manifest, although this is not enforced. Finally, the database table is updated to reflect the output key (ie., a list of output blocks) and the time when the job finished.

While a job is running, mrjobmanager keeps the job table updated with the number of job steps in progress, finished, and remaining. These figures can be retrieved by any Freegol from the batch controller, and displayed to users as a progress indicator.

Order of execution and output assembly is controlled by job step numbers and level numbers. Step numbers begin at zero and are assigned sequentially by mrjobmanager; in the final stage of mrjobmanager the step numbers determine the order in which the job steps’ output fragments are assembled into the final output stream. Level numbers can be used by mr-functions to control the order in which job steps are scheduled: a job step with level *L* will never begin until all job steps with level less than *L* have completed. Each job step can also be given a short input string; this can be used by the mr-function to keep track of which portion of the job each step is expected to compute, in case the job step number itself is not convenient for that purpose.

Typically, a mr-function completes step 0 by reading its input manifest and submitting one new job step for each stream in the manifest. Each of these job steps will read the input stream data, write output data in the form of a stream, store the stream description (one line of the manifest) as a short block in the cache, and report the hash of this short block to the job manager. When all job steps have finished, mrjobmanager looks up all of the individual job step hashes and assembles them, ordered by

job step number, into one final output manifest. This final assembly step is inexpensive because it only involves lists of hashes and filenames; the job manager does not read or write any of the output data itself.

4 Applications and results

4.1 Freegols

We have implemented a few sample applications that demonstrate the warehouse client library and allow us to characterize performance.

The Genomator application is a storage/publication service. It currently allows users to browse and download images from a 300 gigabyte PMAGE data set [17]. In interactive mode, it converts images from TIFF to JPEG format and applies an ImageMagick “normalize” operation to increase contrast. This does not involve any batch processing, and the data set could have easily fit on a single disk; however, implementing the service as a Freegol gave us features like mirrored disks and scalability using existing hardware and staff resources.

The Regol application continuously re-schedules previously completed batch jobs when the cluster is idle, using the same inputs and parameters but substituting the current revision of the relevant mr-function. This helps us notice bugs as they are introduced into the source code repository. It is also a good source of information about performance characteristics of mr-functions, hardware configurations, and resource usage patterns. Regol is deployed on our “templeton” production cluster and is able to view and submit jobs on both clusters.

The Administrator web interface provides a generic job submission and monitoring interface. Users can select a mr-function and revision number, choose an input manifest from the list provided by the storage controller, and specify tunable parameters specific to the mr-function. Each of our clusters has its own Administrator web interface.

4.2 Mr-functions

The mr-functions we have implemented are generally concerned with problems in bioinformatics, specifically low cost DNA sequencing. Here we present some simpler applications that give a rudimentary illustration of the platform. Timing results from two of these can be found in Figure 5.

“mr-pivot-images” reads a manifest with *F* images (one per frame position on a slide, typically about 2000) in each of *C* streams (one per imaging cycle, typically about 75) and outputs a manifest with *C* images in each of *F* streams. The structure of the input data is determined by the image acquisition process; the output struc-

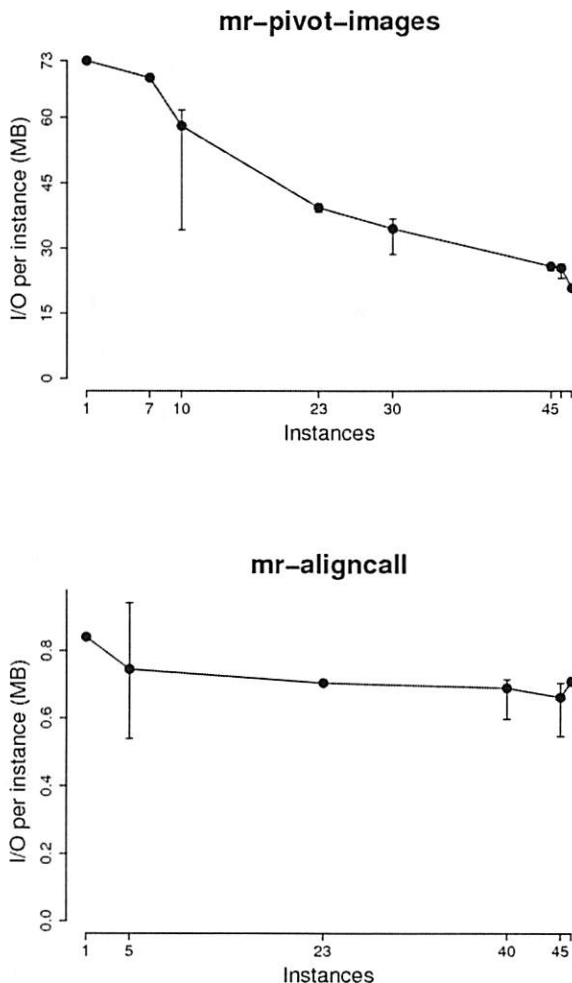


Figure 5: Timing figures from the templeton cluster show the effect of concurrency on per-process I/O speed. Mr-pivot-images shows diminishing returns as more instances are allocated: the data is read from the same set of disks regardless of allocation, and job speed is limited by the disk cache service’s ability to handle many concurrent readers. In contrast, mr-align-call’s performance is nearly linear with larger node allocations.

ture is suitable for image alignment and analysis. This is a relatively inefficient operation because each of F job steps reads pieces from each of C streams. It performs 3.3 terabytes of I/O on our 300 gigabyte input data set. We will certainly want to optimize this if we intend to use it frequently; meanwhile it provides a convenient way to measure a cluster’s performance under heavy I/O load.

“mr-aligncall” reads each stack of images produced by “mr-pivot-images”, analyzes and compares the images according to tunable parameters, and outputs short segments of DNA sequence as strings of A, C, G, and T characters.

“mr-zhash” uncompresses its input (if compressed), computes hashes for individual files, and outputs text files similar to the output of the Linux “md5sum” command line tool. It is useful for determining whether two compressed data sets are equal when decompressed.

“mr-copy” writes a copy of a data set read from a remote cluster (to make subsequent computation faster) or from the local cluster (to verify that all data is readable and passes checksum verification).

As the “mr-pivot-images” example suggests, we have found that the speed of an I/O-limited function is highly dependent on how closely the mr-function’s operation corresponds to the way the data is arranged in the input manifest. Ideally, each step in a batch job reads all of its input data from a single stream, in the exact order in which it is processed. In the case of “mr-aligncall”, this is easy to achieve because 2000 stacks of 75 2 MB images are processed by 2000 independent job steps. In contrast, for subsequent stages of our DNA sequencing applications, we spend much of our effort finding efficient ways to perform operations that are conceptually similar to “mr-pivot-images”. In this sense, the manifest format is an expression of the performance characteristics of the storage service: if we design our workflows to cater to the structure of manifests, then we get the best performance from our system.

4.3 Storage tools

We use two command line tools to move data back and forth between the storage service (or cache) and a Free-gol’s local filesystem. “whput” copies a UNIX filesystem tree to the cache, stores the resulting manifest in the cache, and optionally attaches a name to the manifest via the storage controller. “whget” fetches a manifest from storage, downloads the blocks, computes the MD5 checksums of the individual files, and optionally writes the files to the local filesystem. These tools – and the warehouse client library in general – can be used from any host with access to TCP/IP ports on the warehouse instances, such as an administrator’s workstation.

“whget.cgi” provides a web interface to the con-

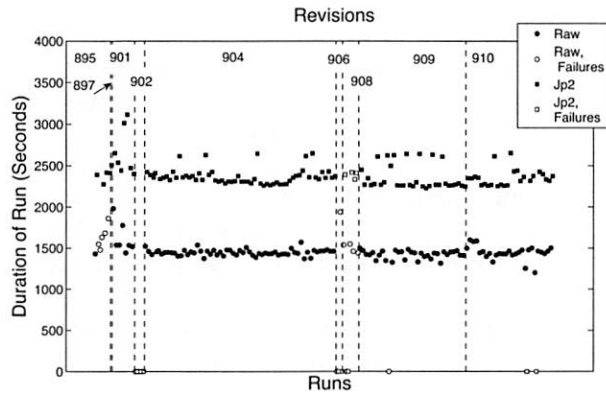


Figure 6: “Regol” helps us notice problems by re-submitting selected jobs when the cluster is idle. This graph shows two different jobs being repeated. Solid shapes indicate successful jobs. Hollow shapes indicate failures. Dashed lines indicate commits to the Subversion repository. If a string of failures begins at a vertical line and ends at another vertical line, it is most likely that the failures were caused by an errant commit.

tents of a manifest. It allows users to view filenames and sizes, click individual files to download them, and download the manifest file itself. Along with Apache’s `mod_rewrite` module, this makes it easy for a developer to selectively publish individual data sets.

4.4 Performance and reliability

In Figure 6 the repeatability of two jobs, “mr-raw” and “mr-jp2”, is illustrated. These jobs are running simultaneously; they perform approximately one terabyte of input and output in each pair of runs. The “jp2” job uses the lossless JPEG2000 format to compress the image data and is CPU intensive. The “raw” job is primarily limited by I/O. By inspection it is clear that the cluster achieves more than 400 MB/s of I/O: 1 terabyte in total, divided by 2500 seconds for the slower job.

To further explore the aggregate I/O and computational capacity of both clusters, we ran a selection of “mr-zhash” cryptographic hash functions concurrently with the “mr-pivot” function described above. The input to mr-zhash is compressed data; in each job, the amount of data processed is over 100 times the amount read from cache. This mixture of computation-intensive and I/O-intensive work was repeated over a 16 hour period, using 42 instances on each cluster. Over the 16 hour period, “mr-zhash” processed 102 TB of uncompressed data (1.5 GB/s on 12 templeton instances, 380 MB/s on 12 uncle instances) while “mr-pivot-images” performed 74 TB of I/O (1 GB/s on 30 templeton instances, 290 MB/s on 30 uncle instances).

In the above test, we ran “mr-zhash” in sets of twelve

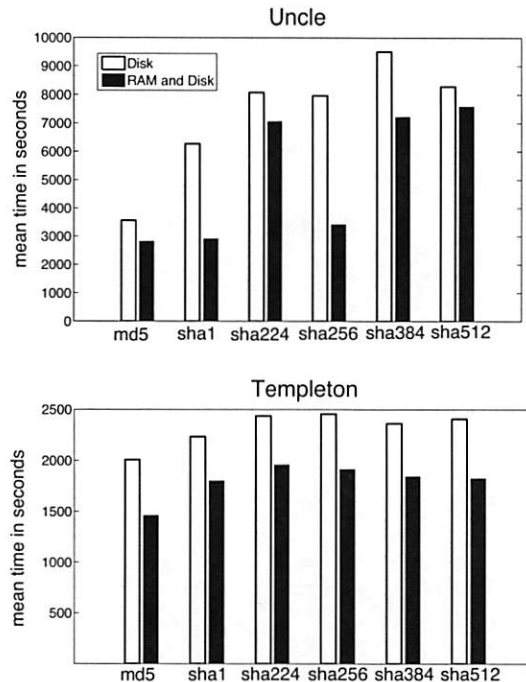


Figure 7: Mean time to read, uncompress, and run various checksum algorithms on 5.5 GB of compressed images, using an allocation of one warehouse instance per job on a busy cluster. Black bars represent jobs with the default client library configuration; hollow bars show the effect of disabling the RAM cache. Execution time is less predictable on uncle: some nodes are much slower than others, and the allocation of nodes to jobs is not random, so we see artifacts in the first graph. Templeton’s hardware is faster and more uniform; this is reflected in the second graph.

concurrent jobs – for each of six hash functions, one job with the RAM cache enabled, and one without. The results are shown in Figure 7.

Many of our design features are responses to lessons we have learned while using our two prototype clusters. For example, because we allocated space for cache on all of our disks before deploying the storage service, the storage service still shares disks with the cache. As a result, the storage service cannot serialize disk accesses. Our disk cache itself was deployed by adding a few disks at a time. This resulted in a poor distribution of data. The manifest used in the “mr-pivot-images” example has 50% of its blocks stored on only 22 disks. As we discussed above, performance suffers when a small number of disks are accessed by a larger number of concurrent processes. It is also noteworthy that even with four concurrent readers on one node, we currently achieve only 75 MB/s of I/O. We have verified with the UNIX `find` program that blocks read in 64 MiB chunks can be

read from the file system at 60 MB/s. We have verified with the `iperf` program that our network can sustain 100 MB/s. Although we have only reached 75% of this limit, rather than the 90% we set out to achieve, we look forward to deploying a new cluster with a properly balanced disk cache and segregated disks for storage. Even if the storage service never surpasses 75 MB/s per node, we are confident that we can achieve 4-8 GB/s of aggregate I/O.

4.5 Utilization

Between November 2007 and March 2008 we completed 460 million seconds of computation (18% utilization) on templeton and 600 million seconds (24% utilization) on uncle. During this time we accumulated 30 TB of data in cache on templeton (consuming 60 TB, 98% of the disk space allocated to the cache service) and 10 TB of data on uncle (consuming 20 TB, 73% of the space allocated). The two clusters consist of new and old hardware costing \$150,000 in total. Annual costs include \$25,000 for floor space, power, cooling, and network service, and about \$50,000 for staff costs.

If we had paid for this CPU time on a per-second basis at the rate charged by Amazon's EC2 for comparable instances – \$0.80 per hour for an extra large instance per node, \$0.20 per hour for two small instances for each of the 36 older uncle nodes – this would cost \$96,000 per year. Storing an average of 20 TB of data for the duration would cost an additional \$36,000 per year, at the Amazon S3 rate of \$0.15 per GB per month.

We can also consider the cost of the time spent copying data between S3 and EC2. Amazon does not specify the usable bandwidth between S3 and EC2, but if we assume that it is a very low 2 Gb/s, it costs \$41 to keep 47 EC2 nodes active while copying 1 TB of data from S3 to EC2. If our 25% utilization rate comes from working on 1 TB of data for one day every four days, the annual transfer cost is less than \$4000 per cluster. This cost is even lower if the available bandwidth is more than 2 Gb/s, which is likely. Therefore, in most cases we expect this transfer cost to be negligible compared to the cost of computation time and storage space.

At these rates, our two clusters will take three years to break even with an Amazon EC2 and S3 implementation. The discrepancy between these figures and those given in section 2.1 is a reflection of the lower number of CPU cores per node in our older hardware, as well as our low hosting costs. Even with this older hardware, a utilization level of 25% is enough to bring the break-even point down to two years.

5 Future Directions

For our projects – and we believe this is true for others too – it is difficult to budget for computation and storage needs. How much we want depends on how much it costs. A platform for universal personalized medicine should permit individuals to form small communities that suit their own needs, while retaining much of the economy of scale available to much larger communities. We believe that this can be achieved by building a highly decentralized global network of Free Factories that allocate underutilized resources through market mechanisms.

Others have explored the possibility of capturing market signals from users and we believe this is an attractive way to allocate resources for our applications in the long term [4, 10, 26]. Since we have control of our architecture from the hardware up, we hope that implementation and experimentation with such mechanisms will be provide an opportunity for fruitful future research.

Finally, in the spirit of free and open source software, we hope others will deploy Free Factories of their own for applications we have never imagined.

6 Acknowledgements

This work was supported by a Center for Excellence in Genome Sciences grant from the National Human Genome Research Institute. We would like to thank our anonymous reviewers for providing numerous insightful comments. The process of making revisions was greatly facilitated by our shepherd, John Wilkes. Art Mann at Silicon Mechanics helped us with our hardware related requests both large and small. Finally, we are in debt to many people who provided data and suggestions, and developed applications for Free Factories.

References

- [1] Amazon.com, Inc. <http://aws.amazon.com/>.
- [2] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS '07*, pages 289–300. ACM Press, 2007.
- [3] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *IEEE Computer*, 39(1):110–112, 2006.
- [4] John Brunelle, Peter Hurst, John Huth, Laura Kang, Chaki Ng, David C. Parkes, Margo Seltzer, Jim Shank, and Saul Youssef. Egg: An extensible and economics-inspired open grid computing platform. In *GECON'06*, Singapore, 2006. World Scientific Publishing.

- [5] Canonical Ltd. <http://ubuntu.com>.
- [6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI'06*, pages 205–218, Seattle, WA, 2006. USENIX Association.
- [7] G. M. Church. The personal genome project. *Molecular Systems Biology*, 1:2005.0030, 2005. <http://personalgenomes.org>.
- [8] Coreboot. <http://coreboot.org>.
- [9] Danga Interactive. <http://danga.com>.
- [10] R.K. Dash, N.R. Jennings, and D.C. Parkes. Computational-mechanism design: a call to arms. *Intelligent Systems*, 18(6):40–47, 2003.
- [11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*, pages 137–150, San Francisco, CA, 2004. USENIX Association.
- [12] Jon G. Elerath and Michael Pecht. Enhanced reliability modeling of RAID storage systems. In *DSN'07*, pages 175–184, 25–28 June 2007.
- [13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP'03*, pages 29–43, New York, NY, 2003. ACM Press.
- [14] GNU Project. <http://gnu.org>.
- [15] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34:34–41, 2005.
- [16] Harvard University. Free Factories source code, 2008. <http://factories.freelogy.org>.
- [17] Jae Bum Kim, Gregory J Porreca, Lei Song, Steven C Greenway, Joshua M Gorham, George M Church, Christine E Seidman, and J. G. Seidman. Polony multiplex analysis of gene expression (PMAGE) in mouse hypertrophic cardiomyopathy. *Science*, 316(5830):1481–1484, Jun 2007.
- [18] Lawrence Livermore National Laboratory. <https://computing.llnl.gov/linux/slurm/>.
- [19] Jeantine E Lunshof, Ruth Chadwick, Daniel B Vorhaus, and George M Church. From genetic privacy to open consent. *Nature Reviews Genetics*, 9:406–411, May 2008.
- [20] Nature Publishing Group. Positively disruptive. *Nature Genetics*, 40(2):119, Feb 2008.
- [21] Opendgear, Inc. <http://opengear.com>.
- [22] OpenVPN, Inc. <http://openvpn.net>.
- [23] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure trends in a large disk drive population. In *FAST'07*, San Jose, CA, 2007. USENIX Association.
- [24] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST'07*. USENIX Association, 2007.
- [25] Jay Shendure, Gregory J Porreca, Nikos B Reppas, Xiaoxia Lin, John P McCutcheon, Abraham M Rosenbaum, Michael D Wang, Kun Zhang, Robi D Mitra, and George M Church. Accurate multiplex polony sequencing of an evolved bacterial genome. *Science*, 309(5741):1728–1732, Sep 2005.
- [26] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent Chun. Why markets could (but don't currently) solve resource allocation problems in systems. In *HotOS-X*, Santa Fe, NM, 2005.
- [27] Software in the Public Interest. <http://debian.org>.
- [28] Alexander Szalay and Jim Gray. 2020 computing: science in an exponential world. *Nature*, 440(7083):413–414, Mar 2006.
- [29] X-Prize Foundation. Archon x-prize for genomics. <http://genomics.xprize.org/>.
- [30] Kun Zhang, Adam C Martiny, Nikos B Reppas, Kerrie W Barry, Joel Malek, Sallie W Chisholm, and George M Church. Sequencing genomes from single cells by polymerase cloning. *Nature Biotechnology*, 24(6):680–686, Jun 2006.
- [31] Kun Zhang, Jun Zhu, Jay Shendure, Gregory J Porreca, John D Aach, Robi D Mitra, and George M Church. Long-range polony haplotyping of individual human chromosome molecules. *Nature Genetics*, 38(3):382–387, Mar 2006.

Wide-Scale Data Stream Management

Dionysios Logothetis and Kenneth Yocum

UCSD Department of Computer Science & Center for Networked Systems

Abstract

This paper describes Mortar, a distributed stream processing platform for building very large queries across federated systems (enterprises, grids, datacenters, testbeds). Nodes in such systems can be queried for distributed debugging, application control and provisioning, anomaly detection, and measurement. We address the primary challenges of managing continuous queries that have thousands of wide-area sources that may periodically be down, disconnected, or overloaded, e.g., multiple data centers filled with cheap PCs, Internet testbeds such as Planetlab, or country-wide sensor installations.

Mortar presents a clean-slate design for best-effort in-network processing. For each query, it builds multiple, static overlays and leverages the union of overlay paths to provide resilient query installation and data routing. Further, a unique data management scheme mitigates the impact of clock skew on distributed stream processing, reducing result latency by a factor of 8, and allows users to specify custom in-network operators that transparently benefit from multipath routing. When compared to a contemporary distributed snapshot querying substrate, Mortar uses a fifth of the bandwidth while providing increased query resolution, responsiveness, and accuracy during failures.

1 Introduction

There is a growing need to monitor, diagnose, and react to data and event streams emitted by wide-scale networked systems. Examples include big-box retailers analyzing retail streams across thousands of locations, real-time weather predictors sourcing hundreds of doppler radars [13], studying network attacks with distributed Internet telescopes [3] or end systems [11, 15], and anomaly detection across wired [20] or wireless network infrastructures [9, 2].

These systems represent a global pool of nodes continuously emitting system and application-specific data streams. In these scenarios, in-network data processing is often necessary as the data streams are too large, too numerous, or the important events within the streams too sparse to pay the cost of bringing the data to a central location. While distributed data processing is important for monitoring large backbone networks [36, 16], where

ISPs collect summary statistics for thousands of network elements, other important applications are emerging. For instance, end-system monitoring specifically leverages the host vantage point as a method for increasing the transparency of network activity in enterprise networks [11] or observing the health of the Internet itself [15]. These environments pose challenges that strongly affect stream processing fidelity, including frequent node and network failures and mis-configured or ill-behaved clock synchronization protocols [24]. This has recently been referred to as the “Internet-Scale Sensing” problem [26].

Mortar provides a platform for instrumenting end hosts, laptop-class devices, and network elements with data stream processing operators. The platform manages the creation and removal of operators, and orchestrates the flow of data between them. Our design goal is to support best-effort data stream processing across these federated systems, specifically providing the ability to manage in-network queries that source tens of thousands of streams. While other data management systems exist, their accuracy is often encumbered by processing all queries over a single dynamic overlay, such as a distributed hash table (DHT) [18, 41, 27]. Our own experience (and that of others [33]) indicates an impedance mismatch between DHT design objectives and in-network stream processing. Even without failures, periodic recovery mechanisms may disrupt the data management layer during route table maintenance, inconsistencies, and route flaps [17].

Mortar incorporates a suite of complementary techniques that provide accurate and timely results during failures. Such an ability facilitates stream processing across federated environments where the set of all nodes in the system is well known, but many nodes may periodically be down, disconnected, or overloaded, e.g., multiple data centers filled with cheap PCs, Internet testbeds such as Planetlab, or city-wide sensor installations [25]. This work complements prior research that has primarily focused on querying distributed structured data sources [18, 14, 27], processing high speed streams [19], managing large numbers of queries [8], or maintaining consistency guarantees during failures [4].

This paper makes the following contributions:

- **Failure-resilient aggregation and query manage-**

ment: Mortar uses the combined connectivity of a static set of overlay trees to achieve resilience to node and network failures. By intelligently building the tree set (Section 3), the overlays are both network aware and exhibit sufficient path diversity to connect most live nodes during failures. Our data routing (Section 3.3) and query recovery protocols (Section 6) ensure that even when 40% of the nodes in a given set are unavailable, the system can successfully query 94% of the remaining nodes.

- **Accurate stream processing in the presence of clock offset:** The lack of clock synchronization, such as the presence of different clock skews (frequencies), can harm result fidelity by changing the relative time reported between nodes (relative clock offset)¹. For distributed stream processing this can increase latency and pollute the final result with values produced at the wrong time. Mortar's *syncless* mechanism (Section 5) replaces traditional timestamps with *ages*, eliminating the effect of clock offset on results and improving result latency by a factor of 8.

- **Multipath routing with duplicate-sensitive operators:** Mortar's *time-division* data management model isolates data processing from data routing, allowing duplicate-free processing in the presence of multipath routing policies. This enables our dynamic tuple striping protocol (Section 3.3), and allows user-defined aggregate operators, without requiring duplicate-insensitive operators or synopses [28].

Extensive experimentation with our Mortar prototype using a wide-area network emulator and Internet-like topologies indicate that it enables accurate best-effort stream processing in wide-scale environments. We compare its performance to a release of SDIMS [41], an aggregating snapshot query system, built on the latest version of FreePastry(2.0.03). Mortar uses 81% less bandwidth with higher monitoring frequency, and is more accurate and responsive during (and after) failures. Additionally, Mortar can operate accurately in environments with high degrees of clock offset, correctly assigning 91% of the values in the system to the right 5-second window, outperforming a commercial, centralized stream processor. Finally, to validate the design of the operator platform, we design a Wi-Fi location sensing query that locates a MAC using three lines of the Mortar Stream Language, leveraging data sourced from 188 sensors throughout a large office building.

2 Design

As a building block for data processing applications, Mortar allows users to deploy continuous queries in federated environments. It is designed to support hundreds of in-network aggregate queries that source up to tens

of thousands of nodes producing data streams at low to medium rates, issuing one to 1000's of records a second. Given the size of such queries, we expect machine failures and disconnections to be common. Thus a key design goal is to provide failure-resilient data stream routing and processing, maximizing result accuracy without sacrificing responsiveness.

We begin by motivating a clean-slate approach for connecting continuous aggregate operators based on static overlays. A goal of this data routing substrate is to capture all constituent data that were reachable during the query's processing window [18, 41, 40], and this work uses result *completeness*, the percentage of peers or nodes whose data are included in the final result, as the primary metric for accuracy. We end this section with how users specify stream-based queries and user-defined operators in Mortar.

2.1 Motivating static overlays

While it is natural to consider a dynamic overlay, such as a distributed hash table (DHT), as the underlying routing substrate, we pursue a clean-slate design for a number of reasons. First, we desire *scoped* queries; only the nodes that provide data should participate in query processing. It is difficult to limit or to specify nodes in the aggregation trees formed by a DHT's routing policy. Second, we can reduce system complexity and overhead by taking advantage of our operating environment, where the addition or removal of nodes is rare. While Mortar peers may become unavailable, they never explicitly "join" or "leave" the system. Further, DHTs are not optimized for tasks such as operator placement [33], and, more importantly, complicated routing table maintenance protocols may produce routing inconsistencies [17].

In contrast, Mortar connects query operators across multiple *static* trees, allowing query writers to explicitly specify the participants or *node set*. Here we take advantage of the relatively stable membership seen in federated systems, which usually have dedicated personnel to address faults. Machines in these environments may temporarily fail, be shutdown for maintenance, or briefly disconnected, but new machines rarely enter or leave the system. The combined connectivity of this tree set not only allows data to flow around failed links and nodes, but also query install and remove commands. This allows users to build queries across the live nodes in their system simply with lists of allocated IP addresses.

This idea builds upon two existing, basic approaches to improving result completeness. Data mirroring, explored by Borealis [4] and Flux [37], runs a copy of the logical query plan across different nodes. Static striping, found in TAG [21], sends $1/n$ of the data up each of n different spanning trees. We compare these ap-

Technique	Benefits
Tree set planning (Section 3)	A <i>primary</i> static overlay tree places the majority of data close to the root operator by clustering network coordinates. Static <i>sibling</i> trees preserve the network awareness of the primary, while exhibiting the path diversity of random trees.
Dynamic tuple striping (Section 3.3)	Route tuples toward root operator while leveraging available paths. Ensures low path length and avoids cycles. Even when 40% of the nodes are unreachable, data from 94% of the remaining nodes is available.
Time-division data partitioning (Section 4)	Isolates tuple processing from tuple routing, allowing multipath tuple routing, and avoiding duplicate data processing.
Syncless operation (Section 5)	Allows accurate stream processing in the presense of relative clock offset, and reduces result latency by a factor of 8.
Pair-wise reconciliation (Section 6)	Leverages combined connectivity of D overlay trees for eventually consistent query installation and removal.

Table 1: A roadmap to the techniques Mortar incorporates.

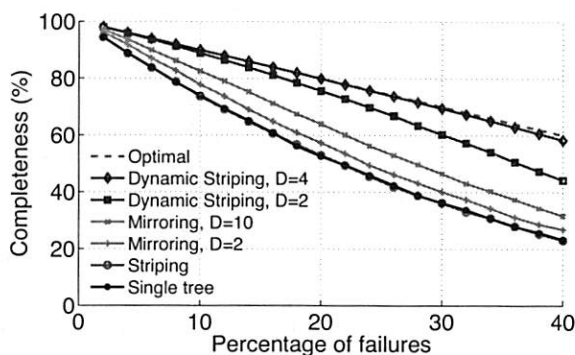


Figure 1: Result completeness under uniformly random failures for mirroring, striping, and dynamic striping. D is the tree set size.

proaches for an aggregate sum operator using a simple simulation. We build random trees (with 10k nodes) of various branching factors, uniformly “fail” random links, and then simply walk the in-memory graph and count the number of nodes that remain connected to the root. Each trial subjects the same tree set to N uniformly random link failures, and we plot the average performance across 400 trials.

Figure 1 shows the ability of mirroring and striping to maintain connections to nodes under different levels of failure, number of trees (D), and a branching factor of 32. Both static options perform poorly. Striping performs no better than a single random tree; many slices of a tree behave, in expectation, as a single random tree. Data mirroring improves resiliency to failure, but at significant cost. When 20% of the links fail, mirroring across 10 trees ($D = 10$) improves consistency by 10% while increasing the bandwidth footprint by an order of magnitude. Obviously, this approach is not scalable.

Instead, we propose dynamic striping, a multipath routing scheme that combines the low overhead of static striping with adaptive overlay routing. Without failures,

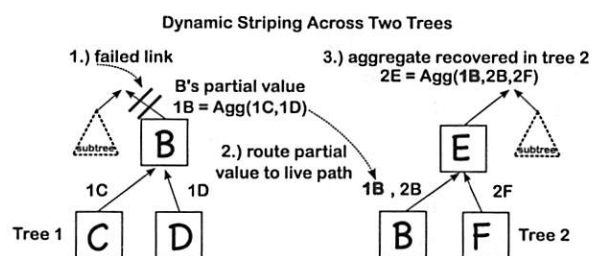


Figure 2: This figure illustrates how dynamic striping avoids failed links (or nodes). Here, after a failed link, node B routes successive partial values to node C on the second tree.

each operator stripes outbound data across each downstream parent in the tree set. When a failure occurs, disconnecting a parent, the operator migrates the stripe to a remaining, live parent. Note that because Mortar is a best-effort system, it does not retransmit data lost due to the failure. Figure 2 illustrates this three step process. This allows nodes to continue to push values towards the root as long as there remains a single live path across the union of upward paths in the tree set. Figure 1, shows that this technique is effective, even with a low number of stripes.

Routing data across a set of static spanning trees for each query sidesteps many of the issues raised by dynamic overlays, but poses new questions. How should one design the tree set so that it is both network aware, but provides a diverse set of overlay paths? How does one route data towards the root while ensuring low path length and avoiding cycles? How should the system prevent duplicate data processing to allow duplicate-sensitive aggregate operators? Meeting these design challenges required us to innovate in a number of areas and led to the development of a suite of complementary techniques (Table 1).

2.2 Queries and in-network operators

Mortar consists of a set of peering processes, any of which may accept, compile, and inject new queries. Each query is defined by its in-network operator type and produces a single, continuous output data stream. It may take as input one or more raw sensor data streams or subscribe to existing data streams to compose complex data processing operations. Users write queries in the Mortar Stream Language².

We require that all operators are non blocking; they may emit results without waiting for input from all sources. An operator's unit of computation is a tuple, an ordered set of data elements. Operators use sliding windows to compute their result, issuing answers that summarize the last x seconds (a time window) or the last x tuples (a tuple window) of a source stream. This is the window *range*; the window *slide* (again in time or tuple count) defines the update frequency (e.g., report the average of the last 20 tuples every 10 tuples).

Mortar provides a simple API to facilitate programming sophisticated in-network operators. Many application scenarios may involve user-defined aggregate functions, like an entropy function to detect anomalous traffic features or a bloom filter for maintaining an index. However, multipath routing schemes often require special duplicate and order-insensitive synopses to implement common aggregate functions [28]. When combined with a duplicate-suppressing network transport protocol, Mortar's data model (Section 4) ensures duplicate-free operation. Thus each in-network operator only needs to provide a *merge* function, that the runtime calls to inject a new tuple into the window, and a *remove* function, that the runtime calls as tuples exit the window. Each function has access to all tuples in the window. This API supports a range of streaming operators, including maps, unions, joins, and a variety of aggregating functions, which are the focus of this work.

3 Planning and using static overlay trees

Mortar's robustness relies on the inherent path diversity in the union of multiple query trees. Our physical dataflow planner arranges aggregate operators into a suitable set of aggregation trees. This means that the system deploys an operator at each source, whether it is a raw sensor stream or the output stream of an existing query. This allows operators to label the tuples according to our data model and reduce the data before crossing the network. The first planning step is to build a network-aware "primary" tree, and then to perform permutations on that tree to derive its siblings. Finally, a routing policy explores available paths while preventing routing cycles and ensuring low-length paths.

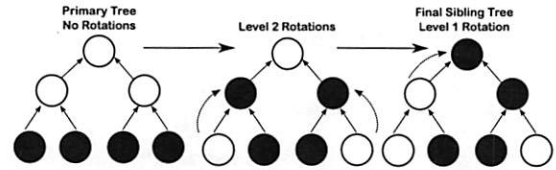


Figure 3: We derive sibling trees from the primary tree through successive random rotations of internal subtrees. This introduces path diversity while retaining some clustering.

3.1 Building the primary tree

Our primary objective is to plan an aggregation tree that places the majority of nodes "close" to the root operator. This allows the root to return answers that reflect the majority of the data quickly. The idea is to minimize the latency between stream sources and their parent operator through recursive data clustering on network coordinates [12]. In network coordinate systems, each peer produces a coordinate whose Euclidean distance from other peers predicts inter-peer latency. Our planning algorithm places operators at the centroids of clusters, avoiding high-latency paths in the top of the query tree.

Mortar treats network coordinates as a data stream, and first establishes a union query to bring a set of coordinates to the node compiling the query. Once at the compiling peer, Mortar invokes a clustering algorithm that builds full trees with a particular branching factor (bf). The recursive procedure takes a root node and the node set. It proceeds by first finding bf clusters, finding the centroids of each cluster, and making each a child of the root. The procedure is then called with each child as the root, and that child's cluster as the node set. The recursion ends when the input node set size is less than or equal to the branching factor.

This design distributes tree building across a small subset of nodes actively used to inject queries. Though the total amount of data brought to the injecting node is relatively small, 10,000 nodes issuing 5-dimensional coordinates results in 0.5MB, the cost is amortized across the compilation of multiple queries. The union query may have a slide on the order of tens of minutes, as latency measurements are relatively stable for those time periods [29].

3.2 Building sibling trees

The key challenge for building sibling trees is retaining the majority of the primary's clustering while providing path diversity. These are competing demands, large changes to the primary will create a less efficient tree.

Our algorithm works in a bottom-up fashion, pushing leaves into the tree to create path diversity. This is im-

portant because interior-node disjoint (IND) trees ensure that failures in the interior of one tree only remove a single node's data in any other. However, complete IND trees would fail to retain the primary trees clustering.

We derive each sibling from the primary tree. The process walks the tree according to a post-order traversal and performs random rotations on each internal node. Figure 3 illustrates the process for a binary tree. Starting at the bottom of the tree, the algorithm ascends to the first internal node and rotates that subtree. The rotation chooses a random child and exchanges it with the current parent. Rotations continue percolating leaves up into the tree until it rotates the root subtree.

While this pushes $\frac{\text{numLeaves}}{bf}$ leaves into the interior of the tree, it doesn't replace all interior nodes. At the same time, it is unlikely that a given leaf node will be rotated into a high position in the tree, upsetting the clustering. Our experimental results (Section 7) confirm this. Note that sibling tree construction makes no explicit effort to increase the underlying path diversity. Doing so is the subject of future work. Finally, an obvious concern is a change in the network coordinates used to plan the primary tree. While we have yet to investigate this in detail, large changes in network coordinates would require query re-deployment.

3.3 Dynamic tuple striping

As operators send tuples towards the root, they must choose a neighboring operator in one of the n trees. For dynamic tuple striping the default policy is to stripe newly created tuples in a round-robin fashion across the trees. However, when a parent becomes disconnected, the operator must choose a new destination. The challenge is to balance the competing goals of exploring the path diversity in the tree set while ensuring progress towards the query root. We explore a staged policy that leverages a simple heartbeat protocol to detect unreachable parents.

Each peer node maintains a list of live parents for all locally installed queries. Each node also maintains a set of nodes from which it expects heartbeats and a set of nodes to which it delivers heartbeats. When the node installs a query, it updates these sets based on the parents and children contained in the query operator. Heartbeats are the primary source of bandwidth overhead in Mortar.

Figure 4 illustrates the intuition behind our scheme. In general the routing policy allows tuples to choose a parent in a given tree only if it moves the tuple closer to the root. To do so, each tuple maintains a list of $\{\text{tree}, \text{level}\}$ pairs that indicates the last level of each tree the tuple visited. Operators consult this list to implement the routing policy. To explain the policy we define four functions. The function $OL(t)$ determines the level occupied by the local operator on tree t . The function $TL(t)$ specifies the

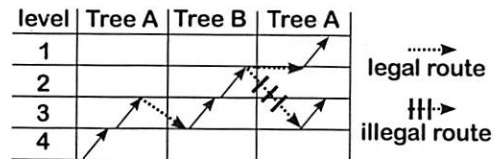


Figure 4: Multipath tuple routing up two trees. To ensure forward progress, tuples route to trees at levels less than the last level they occupied on the tree.

TUPLE ARRIVES ON TREE t	
1	Same tree: Route to $P(t)$
2	Up*: Route to $P(x)$ such that $OL(x) \leq TL(t)$
3	Flex: Route to $P(x)$ such that $OL(x) \leq TL(x)$
4	Flex down: Route to $C(x)$ such that $OL(x) \leq TL(x)$
5	Drop

Figure 5: A staged multipath routing policy. Note that we may choose the tree with the *minimum* level given the constraint.

last level at which the current tuple visited tree t . Functions P and C of t indicate the parent/child of the current node in tree t .

Figure 5 shows the decision process operators use to choose a destination node. Each successive stage allows for more routing freedom, but may also lengthen the path. The first policy attempts to use the same tree on which the tuple arrived. If this parent, $P(t)$, is down, we try "up*", which tries a parent on a different tree, x , that is at least as close to the root as the current tree t . If no such tree can be found, we allow the tuples overlay path to lengthen. The "flex" policy tries to make forward progress on any tree. These first three stages prevent cycles by ensuring that tuples do not re-enter any tree at a level already visited. However, initial experiments showed that they overly constrain the available paths.

Thus, we allow a tuple to descend to the child of a tree chosen by the "flex" policy. This however, does not ensure cycle-free operation, and, when using "flex down", we increment a TTL-down field to limit the possible number of backward steps a tuple can make. When this field is greater than three, stage 4 is no longer available, and the operator drops the tuple. While not shown in Figure 5, we may choose the tree with the minimum level at each stage.

4 Time-division data partitioning

Dynamic tuple striping requires a data model that allows multipath routing. At any moment, a single query may have thousands of tuples in flight across multiple physical dataflows. For example, an aggregate operator participates in each tree (dataflow) simultaneously, and could receive a tuple from any of its children on any tree.

The insight is to allow operators to label tuples with an index that describes the particular processing window to which it belongs. Both time and tuple windows can be uniquely identified by a time range, thus the name *time-division*. With these time-division indices, operators need only inspect the index to know which tuples may be processed in the same window. The scheme is independent of operator type; the data model supports standard operators such as joins, maps, unions, filters, and aggregates. Though this work focuses on in-network aggregates because of their utility, the model also supports content-sensitive operators, those to whom specific tuples must be routed (e.g., a join must see all of the data), by using a deterministic function that maps tuple indices to particular operator replicas.

In many respects, the time-division data model builds upon Borealis' SUnion operator, which uses tuple timestamps to maintain deterministic processing order across operator replicas [4]. However, instead of a single timestamp, it indexes tuples with validity intervals, and defines how to transform those indices as operators process input. Unlike SUnion, the data model underlies all Mortar operators. Its purpose is to allow replicas to process different parts of a stream, not to support a set of consistent, mirrored operator replicas. The data model also differs from previous approaches that parallelize operators by partitioning input data based on its content [6, 37].

The model impacts operator design in two ways. First, an operator computes across a window of raw tuples streamed from the local source, upcalling the merge function for each tuple. This first merge transforms raw tuples into *summary* tuples and attaches an index; we call this merging across time. Note that, if the operator is an aggregation function, then the summary tuple is a partial value. All tuples sent on the network (sent between operators) are summary tuples. An operator's second duty is to merge summary tuples from all its upstream (children) operators. We call this merging across space. The runtime, using the index attached to the summary tuple, calls the same merge function with summary tuples that all belong to the same processing window.

4.1 Indexing summary tuples

Operators create summary tuple indices using two timestamps $[t_b, t_e]$ that indicates a range of time for which the summary is valid. If the window is defined in time, t_b indicates the beginning of the time window and t_e represents the end. If the window is defined across tuples, t_b indicates the arrival time of the first tuple and t_e the arrival time of the last. Thus each summary tuple represents a particular slide of the window across the raw input tuples.

Figure 6 illustrates two nodes creating summary tuples and transmitting them to the root operator. This is

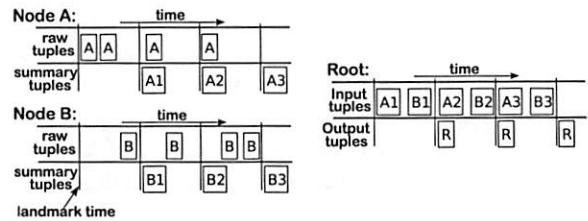


Figure 6: Two nodes creating summary tuples transmit them to the root. Each node (A and B) receives tuples only from its sensor, and labels the summary tuples with a *window index* that uniquely identifies which set of summaries can be merged.

a time window with the range equal to the slide; operators at nodes A and B create indices for each produced summary tuple. This figure illustrates that, for time windows, we can actually use a logical index instead of a time range. The root groups arriving summary tuples with identical indices, upcalls the operator's merge function, and reports a final result R . Here, the root only receives summary tuples.

For time windows, this scheme provides semantics identical to that of a centralized interpretation, assuming synchronized clocks. In our example the root would return identical results had it sourced the data streams directly. This scheme also provides useful semantics for tuple window processing. Instead of calculating over the globally last n received tuples (no matter the source), Mortar's query operators process the last n tuples from each source.

Summaries contain disjoint data for a given time span, and as long as the routing policy and underlying transport avoid duplicates, time-division data partitioning ensures duplicate-free operation. Nodes are now free to route tuples along any available physical query tree, even if it means the summary re-visits a physical node. Note that if a Mortar query consists of content-sensitive operators, upstream operators are constrained in their tuple routing options. In that case, source operators must agree to send the same indices to the same replica.

4.2 The time-space list

An operator may receive summary tuples in any order from upstream operators, and it must merge summary tuples with matching indices. A per-operator time-space (TS) list tracks the current set of *active* indices, indices for which the operator is actively merging arriving summary tuples. The TS list either inserts or removes (evicts) summaries. A TS list is a sorted linked list of summary tuples representing potential final values to be emitted by this operator. Here we assume that each summary tuple is *valid* for its index: $[t_b, t_e]$.

Upon arrival the operator inserts the tuple into the TS

list and merges it with existing summary tuples with overlapping indices. If indices do not overlap, we insert the tuple in order into the list. An exact index match results in the two tuples being merged (calling *merge*). The index of the resulting summary tuple is unchanged. However, when two tuples T_1 and T_2 have partially overlapping indices, the system creates a new tuple, T_3 . T_3 represents the overlapping region, and its value is the result of merging T_1 and T_2 . T_3 's index begins at $Max(T1_{begin}, T2_{begin})$ and ends at $Min(T1_{end}, T2_{end})$. The non-overlapping regions retain their initial values and shrink their intervals to accommodate T_3 . Thus, values are counted only once for any given interval of time.

4.3 Dealing with loss and delay

A common problem in distributed stream processing is telling the difference between sources that have stalled, experienced network delay, or failed. This ambiguity makes it hard for an operator to choose when to output an entry (window) in the TS list. Mortar uses dynamic timeouts to balance the competing demands of result latency and query completeness. The runtime expires entries after a timeout based on the longest delay a tuple experiences on a path to this operator. Each tuple carries an estimate of the time it has taken to reach the current operator ($T.age$), which includes the tuple's residence time at each previous operator. Operators maintain a latency estimate, called *netDist*, using an EWMA of the maximum received sample³. When the first tuple for a particular index arrives, the TS list sets the timeout in proportion to $netDist - T.age$. This is because, by the time tuple T arrives, $T.age$ time has already passed; the most delayed tuple should already be in flight to the operator.

Stalled streams also impact our ability to ascertain summary tuple completeness, and determine how long a tuple-window summary remains valid. To remedy this, operators periodically inject "boundary" tuples when a raw input stream stalls. They are similar in spirit to the boundary tuples used in Borealis [4]. For time windows, boundary tuples are only used to update the tuple's completeness metric (a count of the number of participants); they never carry values. However, boundary tuples play an additional role when maintaining tuple windows. A tuple window only ends when the first non-boundary tuple of the next slide arrives. When a stream stalls, boundary tuples tell downstream operators to extend the previous summary tuple's index, extending the validity interval of the summary.

Finally, Mortar requires that the underlying transport protocol suppress duplicate messages, but otherwise makes few demands of it.

ARRIVAL OF TUPLE T	
1	$O.t.ref \leftarrow O.t.ref + elapsed_time$
2	$index \leftarrow (O.t.ref - T.age) / O.slide$
EVICTION OF TUPLE S	
1	$O.t.ref \leftarrow O.t.ref + elapsed_time$
2	$S.age \leftarrow AVG(T_1.age, \dots, T_n.age)$

Figure 7: Syncless indexing pseudocode.

5 Reducing the impact of clock skew

The performance of distributed stream processing ultimately depends on accurate timekeeping. But assuming synchronized clocks is a well-known problem across large, distributed systems. Even with its wide-spread adoption, NTP may be mis-configured, its ports may be blocked, or it may have limited resolution on heavily loaded nodes [24]. In such cases, differences in clock skew or large clock adjustments can cause substantial differences in reported time between nodes, the relative clock offset. This offset impacts traditional completeness, the percentage of participants included in a window, but also whether the correct tuples are assigned to the window. Here we assess the impact of relative clock offset on *true* completeness, the percentage of correct tuples assigned to a window, tuple dispersion, the distribution of tuples from their true window, and result latency. Our results, presented at the end of this section, show that even mild amounts of offset impact completeness and can increase result latency by a factor of 8.

5.1 Going syncless

This section describes a simple mechanism that improves true completeness, bounds temporal dispersion, and reduces result latency. The *syncless* mechanism requires no explicit synchronization between peers. The intuition is that correct tuple processing depends on the relative passage of time experienced for each tuple. Instead of assigning each tuple a timestamp, we can leverage the *age* of each tuple, $T.age$, a field that represents the number of milliseconds since its inception. Recall from Section 4.3 that this includes operator residence time and network latency. Operators then merge tuples that are alive for similar periods of time at the same index within the time-space list, in the same summary tuple (Section 4.2).

Figure 7 shows the pseudocode used to assign incoming tuples to the correct local index. As Figure 8 illustrates, $O.t.ref$ maintains a relative position in time for each operator, and begins to accumulate time on operator installation. Thus indices are purely local, indicating the set of tuples that should be merged, and may even be negative for some tuples. The evicted summary tuple, S , represents the aggregate of those tuples, and we

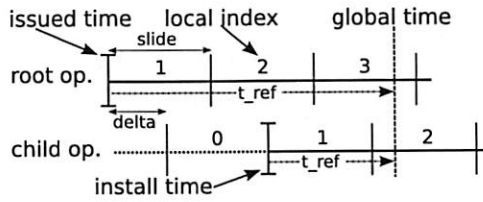


Figure 8: With the syncless mechanism, operators have different install *deltas* relative to the root node.

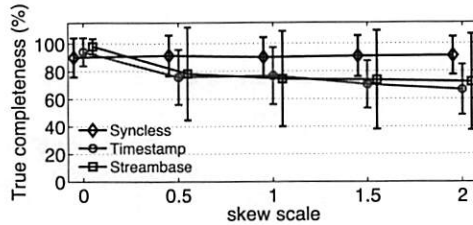


Figure 9: Total completeness for a 5-second window.

set the age of S to the average age of its constituents. This weights the tuple age towards the majority of its constituent data.

One benefit of syncless operation is that it limits tuple dispersion to a tight boundary around the correct window. To see why, first note that operators are not guaranteed to install at the same instant. This results in an install *delta*, $(t_{install} - t_{issued})\%slide$, of the query's install time ($t_{install}$) (seen in Figure 8) relative to the root's install time (t_{issued}). This shifts the local indices for an operator, changing the set of summary tuples merged. Thus, between any two operators, the interpretation of a tuple's age can differ by at most one window. That is, once merged, the tuple may be included in a summary tuple with an average age that places it outside of the true window. We correct for this effect by tracking the *age* of the query installation message, and subtracting *age* from $t_{install}$ on installation. While here the upper bound on tuple dispersion is directly proportional to tree height, dispersion with timestamps is virtually unbounded.

To determine the efficacy of the syncless mechanism we deployed the Mortar prototype over our network emulation testbed, both described in Section 7. Here 439 peers, connected over an Inet-generated network topology, have their clocks set according to a distribution of clock offset observed across Planetlab. 20% of the nodes had an offset greater than half a second, a handful in excess of 3000 seconds. We measure true completeness for an in-network sum, with a five-second window, as we scale the distribution linearly along the x-axis. Each data point is the average of 5 runs. For comparison we plot results from a commercially available centralized stream processor, StreamBase, whose tuple re-order

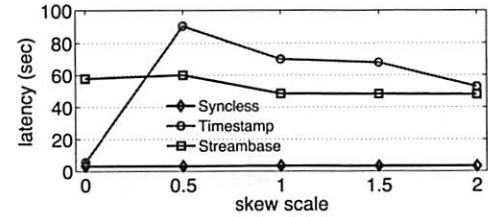


Figure 10: Result latency for a 5-second window.

buffer (BSort operator) we configured to hold 5k tuples.

Figure 9 illustrates true completeness (with std. dev.) while Figure 10 shows result latency for the same experiments. As expected, the timestamp mechanism results in a high-degree of accuracy and low result latency when there is little clock offset. However, at 50% of Planetlab skew, true completeness drops to 75% and result latency for timestamps increases by an order of magnitude. Offset also affects the results from the centralized stream processor, though latency is nearly constant because of the static buffering limit.

In contrast, syncless performance is independent of clock offset⁴, and provides better completeness (averages 91%) than timestamped Mortar or the centralized stream processor at low-levels of skew. Equally important, is that result latency is constant and small (6 seconds).

Large relative clock skew and drift remain potentially problematic. The longer a tuple remains at any node, the more influence a badly skewed clock has on query accuracy. The tuple's residence time is primarily a function of the furthest leaf node in the tree set, and, from our experiments across Inet-generated topologies, is on the order of only a few seconds. Determining this penalty remains future work. However, techniques already exist for predicting the impact of clock skew on one-way network latency measurements [22], and could likely be applied here. Even with these limitations, syncless operation provides substantial benefits in the event NTP is impaired or unavailable.

6 Query persistence

This section discusses how Mortar reliably installs and removes queries across the system. As a best-effort system, Mortar makes no attempt to salvage data that was contained in an operator at the time of node failure. Instead, Mortar uses a pair-wise reconciliation protocol to re-install the same kind of operator, including its type, query-specific arguments, and position in the static primary and sibling aggregation trees, on a recovering node.

Initially, a peer installs (and removes) a query using the primary tree as the basis for an un-reliable multicast. However, because the trees are static, the message must contain the primary and sibling tree topologies. To re-

duce message size and lessen the impact of failed nodes, the peer breaks the tree into n components, and multicasts the query down each component in parallel. While fast, it is unreliable, and the reconciliation mechanism guarantees eventual query installation and removal.

Our protocol draws inspiration from systems such as Bayou [31], but has been streamlined for this domain. In particular, the storage layer guarantees single-writer semantics, avoiding write conflicts, and communication is structured, not random. Like those prior pair-wise reconciliation protocols, the process is eventually consistent.

6.1 Pair-wise reconciliation

Mortar manages queries in a top-down fashion, allowing children who miss install or remove commands to reconcile with parents, and vice versa. The reconciliation protocol leverages the flow of parent-to-child heartbeats in the physical query plan. Periodically, parent-child node pairs exchange summaries describing shared queries. The reconciliation protocol begins when a node receives a summary, a hash (MD5) of relevant queries ordered by name, that disagrees with its local summary. The process is identical for removal operations, but because removals cancel parent-child heartbeats, Mortar overloads tuple arrivals (child-to-parent data flow) for summary comparisons.

First, the two nodes, A and B , exchange their current set of installed queries, $I_{\langle node \rangle}$, and their current set of cached query removals, $R_{\langle node \rangle}$. Each node then performs the same reconciliation process. Each node computes a set of *install candidates* $IC_{\langle node \rangle}$ and *removal candidates* $RC_{\langle node \rangle}$.

$$IC_A = I_B - (I_B \cap I_A) - (I_B \cap R_A)$$

$$RC_A = I_A \cap R_B$$

IC_A represents the set of queries for which A has missed the installation. Additionally, node A removes all queries for which there is a matching remove in R_B . Peers use sequence numbers, issued by the object store, to determine the latest management command for a particular query name. Node B computes IC_B and RC_B similarly. At this point, $I_A == I_B$; reconciliation is complete.

The last step in this process is for the installing peer to re-connect the operator, discovering the parent/child set for each tree in the physical query plan. In this case, the peer contacts the query root, who, acting as a topology server, returns the n parent/child sets. Thus, like planning, Mortar distributes the topology service across all query roots in the system.

7 Evaluation

Mortar has taken a different approach to adaptivity than traditional DHT-based systems that create a single,

dynamic overlay. The ultimate purpose of our techniques is to ensure accurate wide-scale stream processing when node sets contain failed nodes.

Our Java-based Mortar prototype implements the Mortar Stream Language and the data management, syncless, and recovery mechanisms. Each Mortar peer is event driven, leveraging Bamboo's [34] `ASyncCore` class that implements a single-threaded form (based on `SFS/libasync`) of the staged event-driven architecture (SEDA). Other advantages of this low-level integration include `UdpCC`, a congestion-controlled version of UDP, and their implementation of Vivaldi [12] as the source of network coordinates⁵. We use the X-Means data clustering algorithm to perform planning [30]. Beyond the usual in-network operators, the prototype supports a custom trilateration operator for our Wi-Fi location service. Last, aggregate operator results include a completeness field.

We evaluate Mortar primarily on a local-area emulation testbed using ModelNet [39]. A ModelNet emulation provides numerous benefits. First, it tests real, deployable prototypes over unmodified, commodity operating systems and network stacks. A Mortar configuration running over our local cluster requires no code changes to use ModelNet; the primary difference is that, in ModelNet, network traffic is subjected to the bandwidth, delay, and loss constraints of an arbitrary network topology. Running our experiments in this controlled environment allows direct comparison across experiments. 34 physical machines, running Linux 2.6.9 and connected with gigabit Ethernet, multiplex the Mortar peers.

Unless stated otherwise, ModelNet experiments run across an Inet-generated network topology with 34 stub nodes. We uniformly distribute 680 end nodes across those stubs, emulating small node federations. All link capacities are 100 Mbps, the stub-node latency is 1 ms, the stub-stub latency is 2 ms, the stub-transit latency is 10 ms, and the transit-transit latency is 20 ms. The longest delay between any two peers is 104 ms. Each mortar query uses four trees and a branching factor of 16.

7.1 Query installation

Ultimately, query results are only as complete as the operator installation coverage. Reconciliation should install queries across all live, reachable nodes within a node set, even when a significant fraction of the set is down. Here we use a query that sources 680 nodes, but disconnect a random node subset before installation.

Figure 11 shows both the rate and coverage of query installation. Recall that while installation is a multicast operation, it is "chunked", i.e., the installer splits the tree into separate pieces and installs them in parallel. Our experiments use 16 chunks, and with no failures, it takes less than ten seconds to install 680 nodes. We recon-

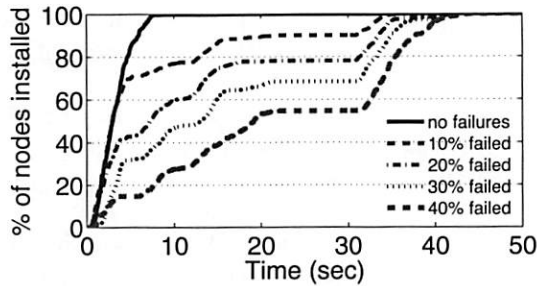


Figure 11: Query installation behavior across 680 nodes with inconsistent node sets.

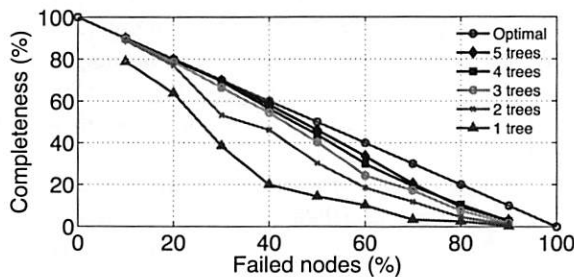


Figure 12: Coverage as a function of the number of trees.

nect all nodes after 30 seconds. Note that reconciliation runs every third heartbeat, i.e., every six seconds, and this results in the slower rate of installation when using reconciliation. However, as predicted by the simulations in Section 2.1, reconciliation installs operators on 54.5% of all nodes even when 40% of nodes are unreachable, resilience similar to that achieved by multipath routing.

7.2 Failure resilience

With the operators successfully installed, the system must now route data from source to query root, avoiding network and node failures. Here our goal is to study how Mortar responds to failures of “last mile” links. Unless mentioned otherwise, these microbenchmarks deploy a sum query that subscribes to a stream at each peer in the system, counting the number of peers. Mortar uses a time window with range and slide equal to one second. A sensor at each system node produces the integer value “1” every second.

7.2.1 The impact of tree set size

Increasing the tree set size improves failure resilience as additional trees add more overlay paths. Here we measure the resiliency additional sibling trees provide and discuss the overhead that comes with it.

Figure 12 plots query completeness as a function of the percentage of disconnected nodes in the system. Here each data point is the average of five runs, each run lasting three minutes. The first thing to note is that with

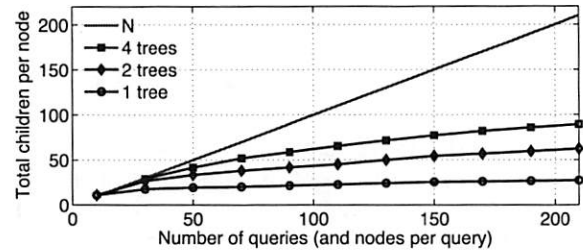


Figure 13: System scaling is directly proportional to the number of unique children at each peer. This plot illustrates sharing across sibling and primary trees as the number of queries increase.

four trees, query results reflect most live nodes, achieving perfect completeness for 10 and 20% failures. For 30% and 40% failures, Mortar’s results include 98% and 94% of the remaining live nodes respectively. This is nearly identical to the results from our simulation and attest to the ability of our sibling tree construction algorithm to create overlay path diversity that approaches that of random trees. Secondly, this appears to be the point of diminishing returns, as five trees provides small additional improvements in connectivity.

Note that each additional tree increases background heartbeat traffic by adding to the number of unique parent-child pairs in the tree set⁶. However, the same heartbeats may be used across the trees in different queries. Figure 13 shows the number of unique children that a given node must heartbeat as a function of the number of queries in the system. Here there is a query for every peer, and that query aggregates over all other nodes. Empirically, overhead scales sub-linearly with both additional queries, and additional siblings per query. In the first case, repeated clusterings on the same coordinate set result in similar primary trees across queries. In the second case, adding a single sibling (2 trees total) roughly doubles the overhead of using a single, primary tree. However, three additional siblings (4 trees total) does not double the overhead of using two trees, but results in a 50% relative increase. This is due to our sibling construction algorithm constraining the possible children a node can have.

7.2.2 Responsiveness

A best-effort system should provide accurate answers in a timely fashion. We first explore the impact of transient “rolling” failures. These time-series experiments disconnect a percentage (10, 20, 30, and 40%) of random nodes for 60 seconds, and then reconnect them. Note that result completeness is identical to that seen in Figure 12 for four trees; here the point is to assess the impact of failure on result latency, completeness, tuple path length, and total network load, the sum of traffic across all links.

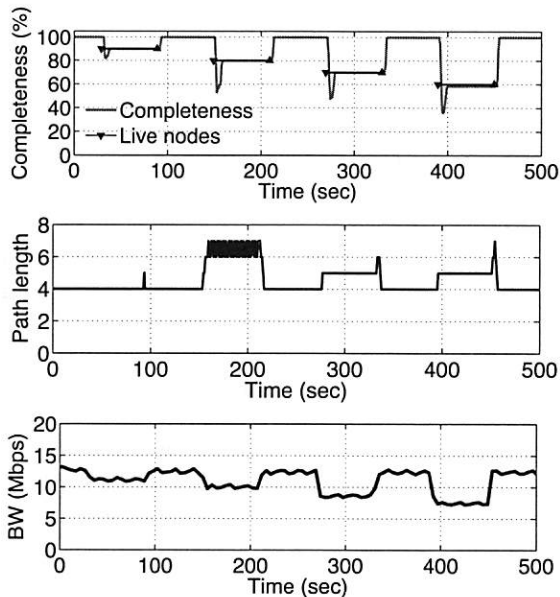


Figure 14: Accuracy and total network load for Mortar during rolling failures of 10,20,30, and 40% of 680 nodes.

Here, result latency is the time between when the result was due and when the root operator reported the value.

Figure 14 shows that Mortar responds quickly to failures, returning stable results on average 7 seconds after each failure. This is a function of our heartbeat period (2 seconds), and appears independent of the number of failures. The system captures the majority of the data with an average result latency of 4.5 seconds. Our branching factor of 16 results in a tree of height 4, which is the path length when there are no failures. Even during 40% failures, the majority of tuples can route around failures with three extra overlay hops. The steady-state network load is 12.5 Mbps (3.4 Mbps of which is heartbeat overhead). As a point of comparison, the same experiment without aggregation incurred twice the network load (26 Mbps).

Finally, while not precisely churn, a query should still be resilient to nodes cycling between reachable and unreachable states. Figure 15 shows results where we randomly disconnect 10% of the nodes. Then, every 10 seconds, we reconnect 5% of the failed nodes and fail an additional, random 5%. Mortar always reconnects all live nodes before the 10 seconds are up. Result latency, network load, and tuple path length are similar to that seen in the rolling failures experiment.

7.2.3 Comparing to a DHT-based system

We compare Mortar to SDIMS [41], an information management system built over the Pastry DHT [35]. We chose SDIMS because of considerable support from its authors, including providing us with a version that uses the latest FreePastry release (2.0.03). This was critical,

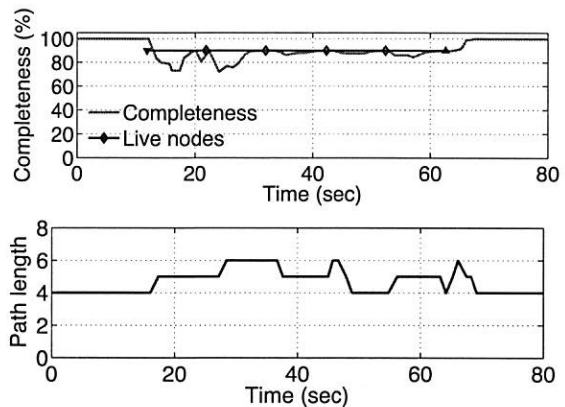


Figure 15: Query accuracy across 680 nodes on an Inet topology during 10% churn.

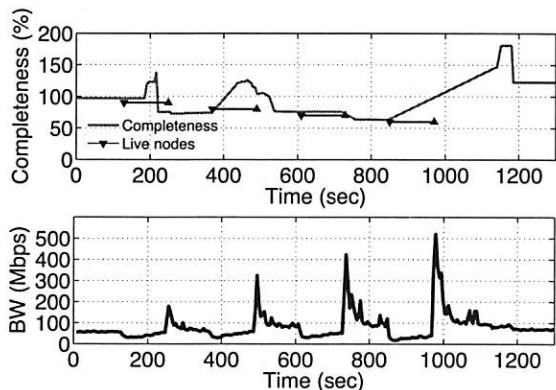


Figure 16: Query accuracy and total network load for SDIMS for 680 nodes. Though we probe five times less often, the steady-state bandwidth is five times greater for the same query.

as it provides routing consistency and explicitly tests for network disconnections⁷. Since SDIMS is a “snapshot” in-network aggregation system, we continuously issue probes to emulate a streamed result.

Figure 16 shows query results and total network load for an SDIMS experiment using 680 peers across the same topology. We fail nodes in an identical fashion, but the down time is 120 instead of 60 seconds. The SDIMS update policy ensures that only the root receives the aggregate value, the ping neighbor period is 20 seconds, the lease period is 30 seconds, leaf maintenance is 10 seconds and route maintenance is 60 seconds. SDIMS nodes publish a value every five seconds and we probe for the result every 5 seconds.

Accurate results at the beginning of the experiment soon give way to highly variable results during even low failure levels. Failures appear to generate over counting as completeness exceeds 100%, hitting almost 180% by the end of the experiment. Probe results remain in-

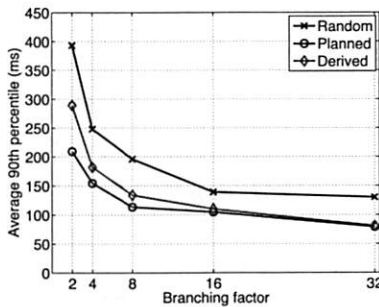


Figure 17: Interconnecting operators across Internet-like topologies. This shows the average latency to the root for the 90th percentile of nodes in the query.

accurate through the end of the experiment, even after all nodes have re-connected. Increasingly large disconnections cause bandwidth spikes as the reactive recovery mechanism engages larger numbers of peers. The steady-state bandwidth usage is 67 Mbps (9 Mbps of which is Pastry overhead); this is 5.3 times as much as Mortar, which produces results with five times the frequency. We hypothesize that the large difference in bandwidth usage is due to a lack of in-network aggregation, as nodes fail to wait before sending tuples to their parents.

As mentioned previously, SDIMS poor accuracy is likely due to its dependence on the underlying DHT for adaptation. Aggressive leaf set and route table maintenance frequencies had little effect.

7.3 Network-aware queries

This section evaluates the ability of our primary and sibling tree building algorithms (physical dataflow planner) to place data within a low-latency horizon around the root operator. Experiments use an aggregate query across 179 randomly chosen nodes over the Inet topology. Vivaldi runs for at least ten rounds before interconnecting operators. We then generate 30 random, primary(planned), and derived (sibling) trees for branching factors (bf) of 2, 4, 8, 16, and 32. For each tree we calculate the latency, across the overlay, from each operator hosting peer to the root operator. This represents the minimum amount of time for a summary tuple from that peer to reach the query root.

Figure 7.3 distills our data to make a quantitative assessment of our planning algorithm. Across each set of 30 graphs, we calculate the average 90th percentile peer-to-root latency. The amount of time the root must wait before it can have a 90% complete value is proportional to this average. First, our recursive cluster planner improves upon random by 30 to 50%. Second, our sibling tree algorithm preserves the majority of this benefit for a range of branching factors.

7.4 The Wi-Fi location service

As a proof of concept we have designed a Wi-Fi device tracking service using the local Jigsaw wireless monitoring system [9] as the source for authentic workloads. Here Wi-Fi “sniffers” create tuples for each captured 802.11a/g frame, containing the relative signal strength indicator (RSSI) measured by the receiver. At each sniffer a select operator filters frames for the target source MAC address. A topk query finds the three “loudest” frames (largest RSSI) received by the sniffers. Finally, a custom trilat operator takes the resulting topK stream and computes a coordinate position based on simple trilateration, given the coordinates of each sniffer⁸.

Unfortunately the Jigsaw sniffers have limited RAM, and cannot accommodate the footprint of our JVM. Instead, we emulate the 188 Wi-Fi network sniffers across the ModelNet testbed; each Mortar peer hosts a “Wi-Fi” sensor that replays the captured frames in real time. The topology is a star with 1 ms links (2 ms one-way delay between each sniffer). Here the primary benefit of physical planning is path diversity, not result latency.

In our experiment, a user circled the four building floor, from the fourth to the first, while downloading a file to their laptop. Figure 18 plots the coordinate stream (x’s); our naive scheme had trouble distinguishing floors, and we plot the points on a single plane. However, this simple query returns the L-shaped path of the user, even distinguishing hallways. Relative to a query that did not allow the TopK to aggregate (bf=188) (but still performing the distributed select), the Mortar query resulted in a 14% decrease in total network load. Without such a selective filter, traditional summary traffic statistics would yield savings similar to those seen in our microbenchmarks.

8 Related work

Mortar’s data model is related to prior work on parallelizing operators, as it allows replicas to process different portions of the same stream. For instance, Flux [37] may partition the input for a hash-join operator using the hash of the join key. Other systems may try to automatically partition the data based on observed statistical properties [6]. However, time-division data partitioning is independent of data content and operator type.

A number of wireless sensor systems employ forms of multipath tuple routing for in-network aggregates. While TAG proposed statically striping data up a DAG [21], two other wireless in-network aggregation protocols, synopsis diffusion [28] and Wildfire [5] allow dynamic multipath routes. Like Mortar, synopsis diffusion de-couples aggregation (for Mortar, merging) and tuple routing, allowing tuples to take different paths towards the root operator. While diffusion allows tuples to be multicast

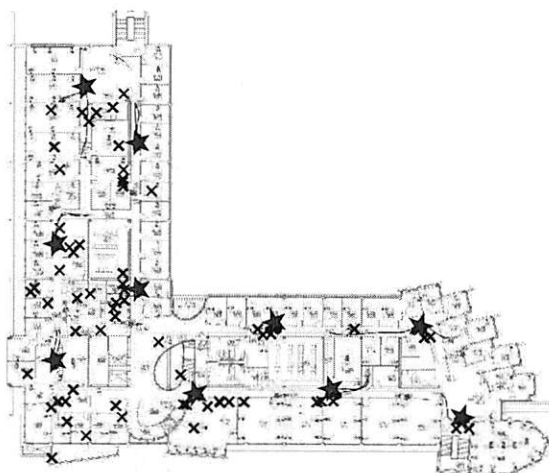


Figure 18: The position of a Wi-Fi user circling the hallways of the UCSD Computer Science department. \times 's represent query results, stars Wi-Fi sensors.

along separate paths, Mortar's data model requires the absence of duplicate summaries. What Mortar offers in its place is a straight-forward operator programming interface. This is in contrast to the special duplicate and order-insensitive operators required of both wireless routing schemes.

The role of Mortar's physical operator mapping is to interconnect operators to create a set of efficient, yet diverse routing paths. Recent work in "network-aware" operator placement tackles a similar problem: placing *unpinned* operators, those that can be mapped to any node in the network, to reduce network load [32, 1]. For example, SBONs [32] use distributed spring relaxation across a cost space combining both network latency and operator bandwidth usage. Our scheme would benefit from their insights in adapting to operator bandwidth usage.

The time management framework proposed in [38] is close in spirit to our syncless mechanism. In that model, a centralized stream processor sources external streams that may have unsynchronized clocks and experience network delays. The system continuously generates per-stream heartbeats that guarantee that no tuple arrives with a timestamp less than τ . However, determining each τ requires the construction of an $n \times n$ (n is the number of streams) matrix whose entries bound the relative clock offset between any two sources. Filling the matrix requires potentially complicated estimations of offset bounds. In contrast, Mortar's syncless mechanism ignores offset altogether, using *ages* to both order tuples and calculate the operator's timeout.

Using multiple trees for increased failure-resiliency has been explored in both overlay and network-layer routing. For example, SplitStream builds a set of interior-node disjoint (IND) trees for the multicast group to bal-

ance load and improve failure resilience. They ensure the trees are IND by leveraging how the Pastry DHT performs routing [7]. Like Mortar, SplitStream sends a separate data stripe down each tree, but the system drops stripe data when encountering failed nodes. An area of future investigation is determining dynamic tuple striping rules for multicasting across a static tree set.

Finally, Motiwala et al. recently proposed a technique, Path Splicing, to improve end-to-end connectivity at the network level [23]. In this scheme, nodes run multiple routing protocol instances to build a set of routing trees; the trees are made distinct by randomly permuting input edge weights. Like Mortar, nodes are free to send packets onto a different tree when a link fails. Their preliminary results show that five trees extracts the majority of the available path diversity, agreeing with ours. While they hypothesize whether such a scheme eliminates the need for dynamic routing in the general case, our experiments indicate that it does for the many-to-one communication patterns in our stream processing scenarios.

9 Conclusion

Mortar presents a clean-slate design for wide-scale stream processing. We find that dynamic striping across multiple static physical dataflows to be a powerful technique, allowing up to 40% of the nodes to fail before severely impacting result streams. Because time-division data partitioning logically separates stream processing and tuple routing, Mortar sidesteps the failure resilience issues that affect current data management systems built over DHT-based overlays. Finally, by reducing the dependence on clock synchronization, Mortar can accurately operate in environments where such mechanisms are mis-configured or do not exist. While it is certain that new issues will arise when deploying a query across a million nodes, Mortar is a significant step towards building a usable Internet-scale sensing system.

References

- [1] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *Proc. of 30th VLDB*, September 2004.
- [2] P. Bahl, R. Chandra, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. Enhancing security of corporate Wi-Fi networks using DAIR. In *Proc. of the Fourth MobiSys*, June 2006.
- [3] M. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, and D. Watson. Data reduction for the scalable automated analysis of distributed darknet traffic. In *Proc. of IMC*, October 2005.
- [4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of ACM SIGMOD*, June 2005.
- [5] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *Proc. of ACM SIGMOD*, June 2004.

- [6] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *Proc. of 31st VLDB*, September 2005.
- [7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proc. of the 19th SOSOP*, October 2003.
- [8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proc. of ACM SIGMOD*, 2000.
- [9] Y.-C. Cheng, J. Bellardo, P. Benko, A. C. Snoeren, G. M. Voelker, and S. Savage. Jigsaw: Solving the puzzle of enterprise 802.11 analysis. In *Proc. of SIGCOMM*, September 2006.
- [10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of CIDR*, January 2003.
- [11] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming network-wide visibility using ubiquitous endsystem monitors. In *Proc. of USENIX Technical Conf.*, May 2006.
- [12] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proc. of SIGCOMM*, August 2004.
- [13] K. K. Droegemeier and Co-Authors. Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. In *20th Conf. on Interactive Info. Processing Systems for Meteorology, Oceanography, and Hydrology*, 2004.
- [14] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An architecture for a world-wide sensor web. 2(4), October/December 2003.
- [15] J. M. Hellerstein, T. Condie, M. Garofalakis, B. T. Loo, P. Maniatis, T. Roscoe, and N. A. Taft. Public health for the Internet (φ). In *Proc. of CIDR*, January 2007.
- [16] L. Huang, X. Nguyen, M. Garofalakis, J. Hellerstein, M. I. Jorran, A. D. Joseph, and N. Taft. Communication-efficient online detection of network-wide anomalies. In *Proc. of IEEE Infocom*, May 2007.
- [17] R. Huebsch, B. Chun, and J. M. Hellerstein. PIER on PlanetLab: Initial experience and open problems. Technical Report IRB-TR-03-043, Intel Corporation, November 2003.
- [18] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER, September 2003.
- [19] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in Gigascope. In *Proc. of 31st VLDB*, September 2005.
- [20] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proc. of SIGCOMM*, August 2004.
- [21] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proc. of the fifth OSDI*, December 2002.
- [22] S. B. Moon, P. Skelly, and D. Towsley. Estimation and removal of clock skew from network delay. In *Proc. of IEEE Infocom*, March 1999.
- [23] M. Motiwala, N. Feamster, and S. Vempala. Path Splicing: Reliable connectivity with rapid recovery. In *Proc. of ACM HotNets-VI*, November 2007.
- [24] S. Muir. The seven deadly sins of distributed systems. In *Proc. of WORLDS*, December 2004.
- [25] R. Murty, A. Gosain, M. Tierney, A. Brody, A. Fahad, J. Bers, and M. Welsh. CitySense: A vision for an urban-scale wireless networking testbed. Technical Report TR-13-07, Harvard University, September 2007.
- [26] R. N. Murty and M. Welsh. Towards a dependable architecture for Internet-scale sensing. In *Second HotDep06*, November 2006.
- [27] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with seaweed. In *Proc. of VLDB*, September 2006.
- [28] S. Nath, P. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. of the 2nd SenSys*, November 2004.
- [29] D. Oppenheimer, B. Chun, D. A. Patterson, A. Snoeren, and A. Vahdat. Service placement in a shared wide-area platform. In *Proc. of USENIX Technical Conf.*, June 2006.
- [30] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proc. 17th ICML*, 2000.
- [31] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of 16th SOSOP*, October 1997.
- [32] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. of ICDE*, April 2006.
- [33] P. Pietzuch, J. Shneidman, J. Ledlie, M. Welsh, M. Seltzer, and M. Roussopoulos. Evaluating DHT service placement in stream-based overlays. In *Proc. of IPTPS*, February 2005.
- [34] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of USENIX Technical Conf.*, June 2004.
- [35] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, pages 329–350, November 2001.
- [36] V. Sekar, N. Duffield, O. Spatscheck, K. van der Merwe, and H. Zhang. LADS: Large-scale automated DDoS detection system. In *Proc. of USENIX Technical Conf.*, May 2006.
- [37] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault tolerant, parallel dataflows. In *Proc. of ACM SIGMOD*, June 2004.
- [38] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of PODS*, June 2004.
- [39] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. of OSDI*, December 2002.
- [40] R. VanRenesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, 2003.
- [41] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc. of SIGCOMM*, September 2004.

Notes

¹We borrow definitions of skew (differences in clock frequency) and offset (difference in reported time) from the network measurement community [22].

²It is a text-based version of the “boxes and arrows” query specification approach [10, 4].

³ $\alpha = 10\%$ worked well in practice.

⁴What variations exist are due to the random placement of offset across the nodes for each test.

⁵Experiments use 3-dimensional coordinates.

⁶In the worst case each tree adds $O(N)$ pairs globally and $O(bf)$ at involved peers. N is the node set size and bf the branching factor.

⁷Experiments with PIER [18] showed that it was badly affected by the dynamism in the Bamboo DHT’s [34] periodic recovery protocols. The PIER authors have made similar observations [17].

⁸We are not innovating here; more advanced methods exist but could use similar queries.

Experiences With Client-based Speculative Remote Display

John R. Lange* Peter A. Dinda* Samuel Rossoff†
{jarusl,pdinda}@cs.northwestern.edu, srossoff@csc.uvic.ca
*Northwestern University †University of Victoria

Abstract

We propose an approach to remote display systems in which the client predicts the screen update events that the server will send and applies them to the screen immediately, thus eliminating the network round-trip time and making the system more responsive in a wide-area or high loss environment. Incorrectly predicted events are undone when the actual events arrive from the server. The approach requires no server or protocol changes, and thus can work with existing systems. Since it is core to the feasibility of such a speculative remote display system, we study the predictability of the events that occur under typical workloads in two extant systems, Windows Remote Desktop and VNC. We find that simple, state-limited Markov models are often able to correctly predict the next event. Based on these results, we design, implement, and evaluate a speculative remote display extension to the VNC client. In our implementation, the end user can trade off between the responsiveness of the display and the level of temporarily displayed incorrect predictions. We evaluate VNC/SRD with two user studies. We conclude by describing design alternatives.

1 Introduction

Remote display systems allow a distant user to control a computer or application with a graphical user interface. While this technology dates back to the 1980s and the X Window System [18], it has only recently become widely deployed through the success of VNC [17] and Microsoft's inclusion of the Remote Desktop Protocol (RDP) in the mainstream release of Windows. Remote display systems are also key components in thin-client computing [9, 19]. In this paper, we particularly consider the RDP and VNC protocols.

Increasingly, remote display systems are being used over wide-area networks where round-trip latencies are inherently higher than those in local-area networks, and have far greater variance [16, 7]. These higher latencies dramatically reduce the utility of remote display systems for end-users, making the remote computer seem choppy, slow, and hard to control.

The client and server in a remote display system communicate through two independent event streams. User events (keystrokes, mouse movements and clicks, etc) flow from the client to the server, while screen events (graphics primitives) flow from the server to the client. While neither VNC nor RDP require that the two event streams be synchronized, they are in fact synchronized through the user who will frequently wait for the effects of his actions to be shown in his display. The user is thus subject to the round-trip time of the network path and perceives the high mean latency and variance of the path as slowness and jitter in the display. While important recent work, both in academia [1], and industry [12], have made remote display systems more efficient, the client/server round-trip is fundamentally limited by the physics.

We propose *speculative remote display* as a way of alleviating this problem. Speculation has been proposed as a way to reduce the user-experienced latency in other systems contexts [24, 13]. In this paper, we focus on an approach to speculative remote display that requires no server or protocol changes, and thus can be layered over existing systems, including those previously mentioned. We assume that the client has spare cycles. The key idea is for the client to predict future screen events from the history of past screen and user events and execute these screen updates immediately. As the actual server-supplied screen events arrive, they are compared against the predicted events that have already been speculatively executed. If there is a difference, the client rolls back (undos) the effects of the erroneously predicted events. In effect, the client is always executing

Effort funded by the National Science Foundation under award CNS-0720691. Lange was funded by a Symantec Research Labs Fellowship.

a nested transaction on the frame buffer, and the updates from the server indirectly provide commit and abort notifications. Provided that (1) the event streams are sufficiently predictable, and (2) users are sufficiently tolerant of rollbacks, speculative remote display has the potential to make wide-area remote display more usable.

A client-based speculative remote display system requires significant client-side changes, and the selection of an appropriate predictor. The client must be modified so that it can interface with a predictor, validate predictions, and be able to roll back predicted display changes when they prove to be wrong. The difficulty of these modifications is protocol-dependent. RDP clients are considerably harder to modify than VNC clients because the RDP protocol is semantically much richer, and because RDP requires that the client maintain state beyond the framebuffer and cache, namely a Windows-like device context. For this reason, we have designed, implemented, and evaluated a speculative client for VNC to begin exploring speculative remote display. However, we have studied the predictability user and screen event traffic in both RDP and VNC.

There are numerous approaches that one could take to predicting the future screen events from the past user and screen events. Our prototype system modularizes the predictor. The initial predictor we use in our work is simple state-limited k -th order Markov model that uses the events themselves as symbols. We describe this predictor in more detail in Section 2.

Using this predictor, we examined the prospects for client-based speculative remote display by studying the predictability of traces collected during a study of users using instrumented clients for Microsoft's RDP and for VNC. In both cases, our users interacted with commonplace Windows applications. For RDP, our simple Markov models are very successful in predicting the next screen event from previous screen and user events. Not only can we generate a prediction for most events, the predictions are usually right. For VNC, our models are less successful. We generate predictions for a smaller fraction of the events, but the predictions are usually right. However, the rate of VNC screen events is also much lower than that of RDP. We describe our predictability analysis in detail in Section 3.

We designed and implemented client-based speculative remote display within the VNC system. Our system is described in detail in Section 4. There are several design alternatives possible, and we explain the implications of each and how we arrived at our current design. Our system's user interface lets the naive user trade off between the effective reduction in round trip time and the degree of display artifacts appearing on the screen due to incorrect predictions.

Because our system essentially trades off the effective

responsiveness¹ and correctness² of the display, it is essential that we evaluate whether such a remote display approach is usable, whether users can select a reasonable tradeoff point using our interface, and to what extent we can improve the user experience. To do so, we designed and undertook user studies, described in Section 5.

We are also considering approaches to speculative remote display that introduce server and protocol changes, as well as client changes. We comment on these approaches and on the nature of protocols that seem to be best suited for speculative remote display in Section 6 and conclude in Section 7.

Other related work

While prediction-based latency hiding has been used in many domains, to the best of our knowledge, ours is the first application of this idea to remote display.

A related domain is multiplayer networked games where each client is responsible for maintaining a private representation of the game state which is updated by local and remote game events. Prediction-based techniques to hide the latency of remote events include dead reckoning [10], and systems that rely on knowledge of game AI implementation to speculatively perform non-critical actions [15]. In contrast to remote display events, game events are both application-specific, and at a much higher semantic level.

There is considerable interest in predicting user behavior within user interfaces. Examples include predicting Unix command-lines [4], and predicting the shapes that a user is drawing in a CAD tool [20]. While our system does predict in part based on past user events, we predict future *system actions*, not future *user actions*.

Our work also bears some resemblance to the bountiful work in prefetching for web [14] and file access [8]. However, speculative remote display is not prefetch as it predicts the *contents* and not the request.

Finally our work can be compared to speculative execution as being explored in the context of both operating systems [22] and distributed systems [6, 13, 23]. In contrast, our focus is on speculation for a specific system service, remote display, not for the general problem of speculative execution.

2 Predictor

Our predictor is a k -th order Markov model. The symbols that the Markov model operates on are the user or

¹The user sets a round-trip-time *goal*. The success in reaching the goal depends on the predictability of the traffic.

²It is important to note that all incorrectness in the output of our system is *temporary*. Display artifacts caused by incorrect predictions are *repaired* as quickly as possible.

system events as supplied as human-readable strings. For events that involve bitmap content, we hash the content, maintain a mapping from the hash back to the actual content, and use the hash as part of the symbol. A typical event contains the type of event (e.g., mouse movement) and its parameters (e.g., (x, y) coordinates). A state is the simple concatenation of the last k symbols.

Of course, given this simple scheme, the bound on the state space size of a model is $O(n^{k+1})$, where n is the number of distinct input symbols. Furthermore, because we include parameters in the symbols, n can potentially be astronomical. For this reason, our implementation can constrain the number of states to be between an upper and lower limit, keeping the most visited states and garbage collecting the rest when the upper limit is reached.

Our implementation supports continuous model fitting and prediction. That is, it can operate on a stream of symbols, updating the model on each new symbol as well as supplying a prediction of the symbol that is most likely to occur next. If there is insufficient information (e.g., we are in a state which currently has no outgoing arcs), then the predictor does not attempt to predict the next state.

At any point in time, the predictor can be asked for an arbitrary number of subsequent events. It will return as many subsequent events as possible, up to that bound. It can return fewer events, however, as it is possible that given the state transitions that have been seen up to the present time, insufficient data may exist to compute an empirical distribution over the states for some number of steps ahead. As a simple example, if we are presently in a state that has not been seen before, no next state can be computed.

Our predictor is implemented in Perl. The core consists of a library of approximately 600 source lines of code. We can use this library online, in which case an additional 250 lines of Perl are included. This online predictor consumes a stream of symbols (and requests for predictions) and produces a stream of vectors of predicted symbols.

Although the choice of Perl may seem odd from a performance perspective, it is important to note that a tool like this spends most of its time in hash table lookups, and that most Markov model update and prediction operations can be implemented with high level constructs (e.g., *map*). Because of this high level representation, and the fact that Perl's hash implementation is very efficient, we believe that our predictor implementation is not much slower than one written in a low-level language.

We note that other predictors could be applied in this domain, for example, Prediction by Partial Matching (PPM) [3]. As we discuss in Section 4.4, our system allows predictors to be easily plugged in.

3 Predictability Study

A key requirement for speculative remote display is that there be, in practice, a considerable degree of predictability in the user and screen event streams. In the following, we apply the simple Markov model described in Section 2 to the problem of predicting the next screen event from past screen events and from both past screen events and past user events. The system we describe in Section 4 can be configured either way, and our studies in Section 5 use past screen and user events to predict future screen events. We study both the RDP protocol and the VNC protocol. We also briefly comment on predicting future user events from past user events.

3.1 Windows Remote Desktop

We first consider the predictability of screen events in the RDP protocol.

3.1.1 Traces

To collect trace data to evaluate our predictor, we instrumented the rdesktop [2] open source RDP client (version 1.4.1) so that it non-intrusively records all user and screen events to files. We then created an experimental testbed consisting of 2 PCs (P4, 2 GHz, 512 MB, 19" LCD display, Windows XP SP2 (server) or Red Hat Linux EL4.4 (client)) connected via a private 100 mbit network. Notice that this is an ideal remote desktop configuration—network latency and jitter are minimized. To collect a trace, the user sat at one PC and used rdesktop (at 1280x1024 resolution and 24 bit color) to use the other PC. The user performed the following tasks:

- Acclimatization. (5 minutes)
- Word processing with Microsoft Word 2003. The user spent 15 minutes recreating a supplied document.
- Presentation creation with Microsoft Powerpoint 2003. The user spent 15 minutes recreating a supplied document with considerable drawing required.
- Web browsing using Firefox 2.0. For 15 minutes, the user visited a news web site, read an article, and then conducted web searches on its topic in another window. (15 minutes).

Five users participated. They included graduate students and faculty in the EECS Department at Northwestern. The user traces contain 47 to 77 thousand events, while the screen traces contain 712 thousand to about one million events.

3.1.2 Results

We ran the screen and user traces generated by each user through our online Markov predictor, varying the order

of the model and the upper and lower limits on the number of states in the model. For screen→screen prediction, we simply use the screen trace. For screen+user→screen, we merged the screen and user traces by timestamp, and discarded predictions of user events. For each combination of trace, order, and limits, the predictor started with no information at the beginning of the trace and formed its model progressively as it saw the input symbols. This is identical to how a predictor in a speculative remote display system would operate.

The graphs in Figure 1 show the percentage of prediction attempts that are successful, as a function of the order of the model. In (a)–(c), we consider screen→screen, with a progressive limitation on the number of states permitted, while (d)–(f) considers screen+user→screen with the same progressive limitation. Each curve corresponds to a particular user. The key point is that with an practical 1000–2000 state model ((b) and (e)), over 90% of attempts to predict the next screen event are successful. If we have seen any transitions out of a state before (i.e., if we have seen the state before), we almost always predict the next state correctly. Note including user events in the input to the predictor is necessary to be able to eliminate the user event to screen event round trip noted in the introduction.

The graphs of Figure 2 correspond exactly to those of the previous figure with the exception being that we are plotting the percentage of all events that are successfully predicted. Remarkably, with our 1000–2000 state model, we are still able to correctly predict over 60% of screen events.

The upshot of Figures 1 and 2 is that we have found that both RDP screen and user events are surprisingly predictable with even an extremely naive predictor.

We are also able to predict 6–8% of user events from past user events, with 60–80% of such predictions being accurate.

3.2 VNC

We also evaluated the predictability of screen events in the VNC protocol. While VNC's user event model is similar to that of RDP (key and mouse events), it is important to note that VNC's screen event model is much simpler than that of RDP. VNC's model essentially boils down to drawing rectangles of bitmaps to the framebuffer, where the bitmap content is supplied in the message. Many encodings for the bitmap data are possible. Note that in RDP, only about 20% of messages are of this nature.

3.2.1 Traces

To collect trace data to evaluate our predictor, we instrumented a VNC [17] client, specifically Real VNC Viewer 4.1.2 for the X window system. We ran the modified viewer on a Linux-based client computer and connected to a Windows XP SP2 server via a private network. Further details about the computer, screen, and network configuration are given in Section 5.2.

Five users participated. They included undergraduates and graduate students in the Computer Science Department at the University of Victoria. The users carried out the same tasks as described earlier. The user traces contain 12 to 24 thousand events, while the screen traces contain 8.5 to 17 thousand events.

Notice that the VNC traces include about 1/2 as many user events and 2 orders of magnitude fewer screen events than the RDP traces. The latter is due to the semantically poorer content of the VNC screen updates. In VNC many drawing commands at the GDI level will be bundled together into a small number of bitmap rectangle updates, while in RDP, the updates sent to the client correspond much more closely to the GDI drawing commands.

3.2.2 Results

Using a presentation that mirrors that of Section 3.1.2, we now show our analysis of the predictability of the VNC traces. We use the same range of model orders (1 to 8) and state size limits (100-200, 1000-2000, unlimited) as before.

The graphs in Figure 3 show the percentage of prediction attempts that are successful, as a function of the order of the model. Figures 3(a)–(c) are for screen→screen, with progressive limitation on the number of states permitted, while Figure 3(d)–(f) are for screen+user→screen with the same progressive limitation. Each curve corresponds to a particular user. Similar to the RDP traces (the comparable figure is Figure 1), we see that, provided a sufficiently large model (again 1000-2000 states), when we *can* predict the next screen event, we are usually successful, although not quite as successful as in RDP.

The graphs of Figure 4 correspond exactly to those of the previous figure, except that here we plot the percentage of all events that are successfully predicted. The comparable figure for RDP is Figure 2. For screen→screen (Figure 4(a)–(c)), the overall predictability of VNC is much worse than that of RDP. Adding user events to the input (Figure 4(d)–(f)), slightly increases the overall predictability. While we predict just as accurately when we do make a prediction, we have far fewer instances in which we can make a prediction. This difference seems to derive from two factors. First, we see far fewer screen events in VNC than we see in RDP. All

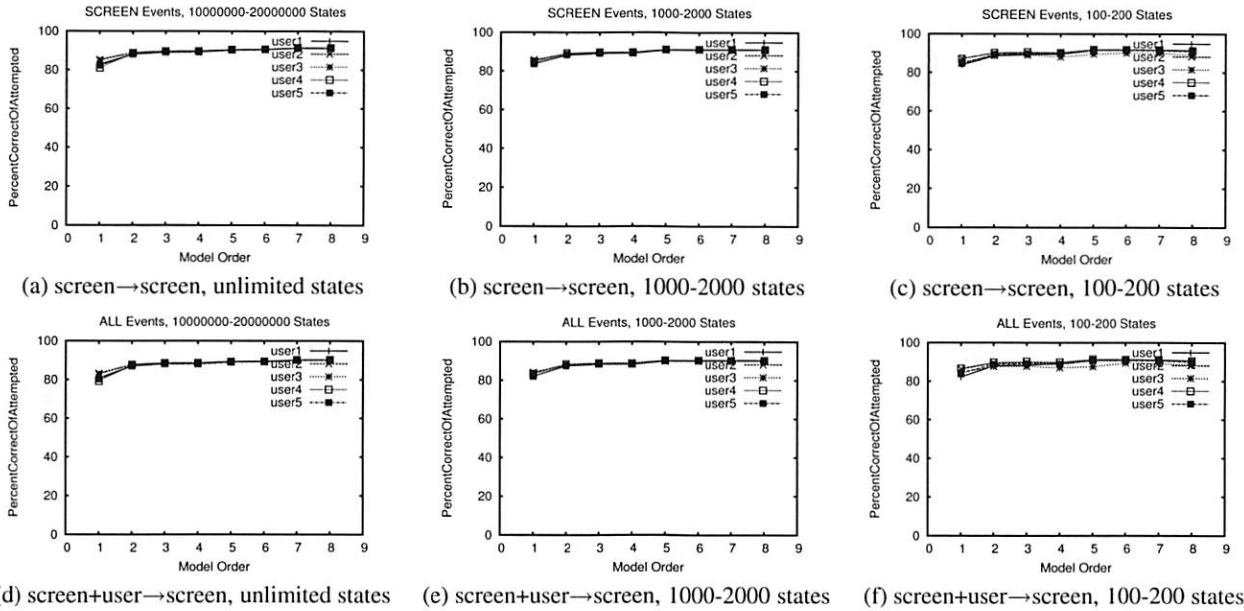


Figure 1: Percentage of prediction attempts that are correct. (RDP study)

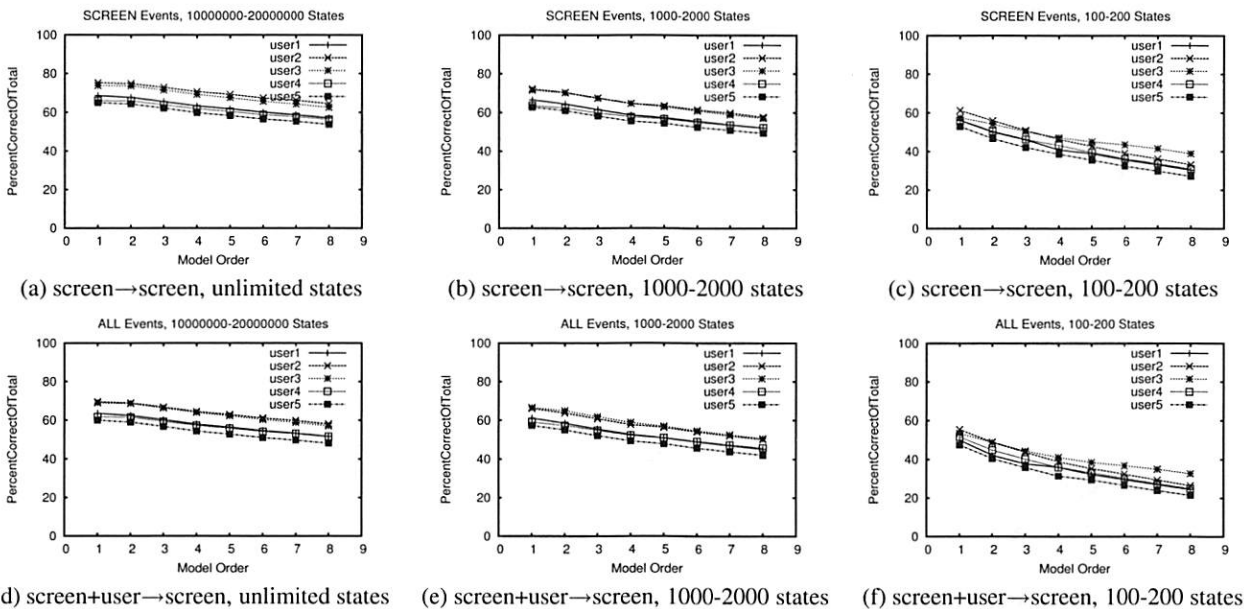


Figure 2: Percentage of all events that are correctly predicted. (RDP study)

other things being equal, the model simply cannot grow as robust with so much less data. Second, because a VNC screen event contains many more GDI-level drawing commands, compared to an RDP screen event, the likelihood that two events will share the same sequence of GDI-level events (and thus the same bitmap) is lower.

Despite the lower overall predictability of screen events in the VNC protocol, it is important to point out that we do, in fact, see some predictability. With appro-

prate choices for the model order and the state size limits, 2-7% of screen events can be accurately predicted. With enough states, and most predictions are accurate.

For predicting user events from past user events, we would the overall predictability in VNC is slightly higher than that in RDP.

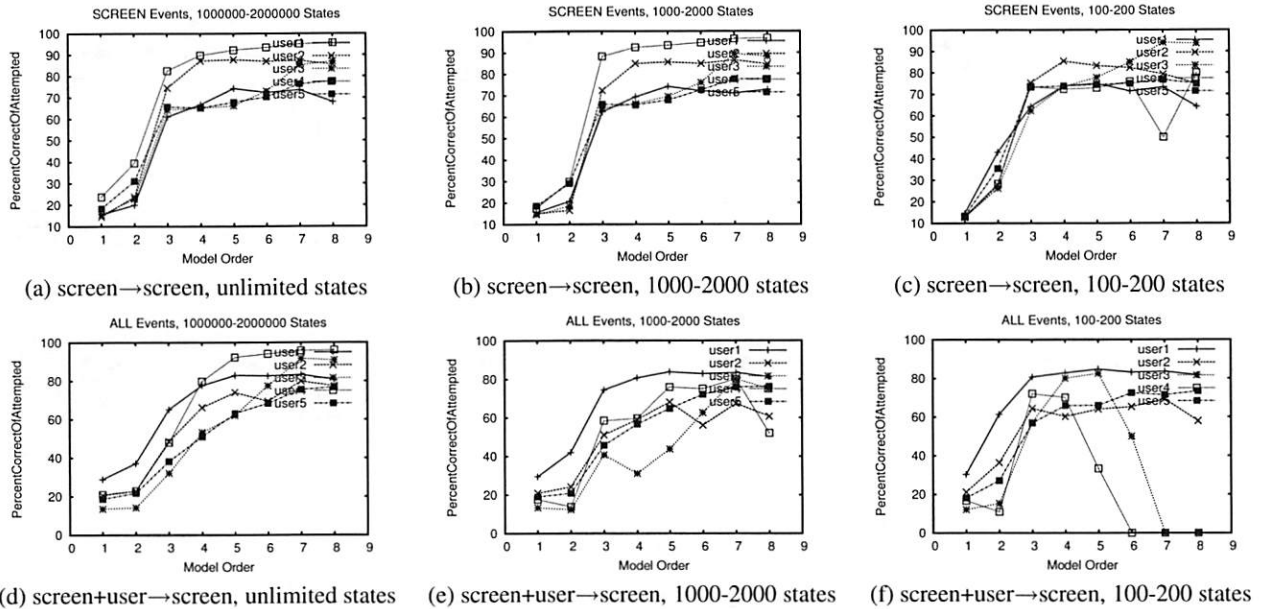


Figure 3: Percentage of prediction attempts that are correct. (VNC study)

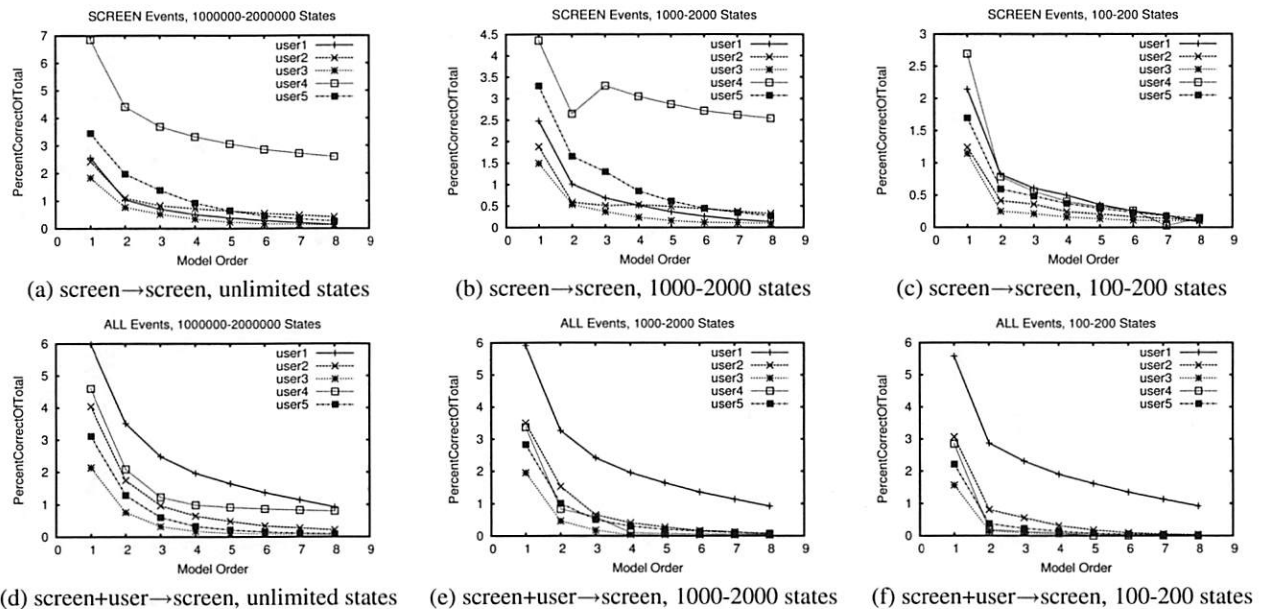


Figure 4: Percentage of all events that are correctly predicted. (VNC study)

4 Design and implementation of VNC/SRD

We now describe the overall design and implementation of a version of VNC that incorporates client-based speculative remote display. VNC/SRD consists of the predictor described in Section 3 and extensions to the Real VNC Viewer 4.1.2 code base. Our extensions are implemented in ~ 2300 lines of C++. No protocol changes are made. The VNC/SRD client connects to an unmodified VNC server.

Figure 5 illustrates the design of VNC/SRD. The prediction framework that we have incorporated into VNC intercepts user and screen events, maps them into appropriate *event signatures*, feeds the signatures to the external predictor, requests predictions as needed, maps predicted event signatures back to screen events, and applies the screen events to change the display. As actual screen events arrive from the server, it also reconciles them with predictions, and repairs the display as needed.

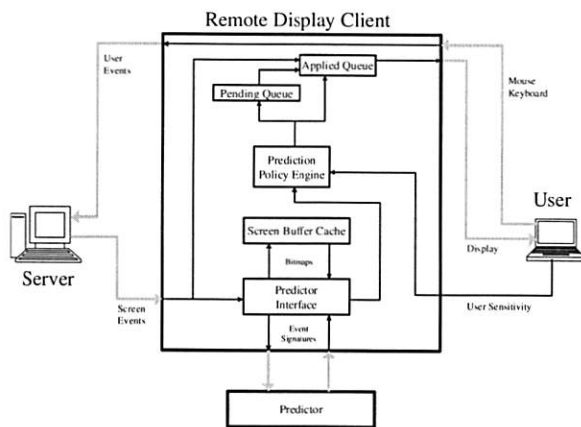


Figure 5: Design of VNC/SRD.

4.1 Why not RDP/SRD?

Given the results of Section 3, which show that RDP screen events are generally more predictable than VNC screen events, the reader may wonder why we chose to work in the context of VNC. While we plan an RDP/SRD client based on the `rdesktop` code base, we chose to design and implement VNC/SRD first because the protocol is much simpler, and requires very little client-side state. The VNC protocol consists of 4 different events, while RDP contains many more. The on-screen result of a VNC event depends only on the event and the current display, while an RDP event's result depends additionally on separate state (essentially a device context), and it causes side effects on that state. The path from event to screen in VNC Viewer is very straightforward, while the `rdesktop` path is more complex. Furthermore, it is important to note that while we are generally more successful at predicting the next screen event in RDP, individual VNC screen events occur at a much lower rate and generally affect far more pixels. Finally, we note that if client-based speculative remote display is acceptable to users at some prediction error rate, it is surely even better at a lower one.

4.2 Event interception

Intercepting VNC events requires a number of changes to the VNC event handling architecture. By default, VNC streams events through the display client with minimal processing. We changed the implementation to buffer full events before applying them. Buffering the events is necessary for us to validate events against predictions and also to simplify the generation of event signatures. VNC user events are trivially buffered since they consist of only a few parameters—we simply capture them as they are written to the network. VNC screen events, however, are streamed directly to one of a set of decoders

which then streams the decoded image directly to the user's framebuffer. To buffer these screen events we redirect the decoder's output to an internal framebuffer that is dynamically created for the incoming event. The internal framebuffer is then forwarded on to be rendered once the event has been fully received.

4.3 Event/prediction mapping

There is an inherent tradeoff in how the system maps between the semantic information for an event and the symbols that the predictor operates on. This mapping must be maintained by either the predictor or the client. Consider a design that optimizes for the run-time costs of the predictor. In this case the symbols sent to the predictor would need to be as compact as possible. This would require the client to maintain mappings for all the events that it receives, even if many of the mappings will never be used. If we optimized for the run-time cost of the client, we would move the responsibility for mapping to the predictor. This would require transferring all of the event's semantic information to the predictor. In the worst case, this would result in transporting full screen updates to the predictor.

In our implementation, we *split* the mapping between the client and the predictor. For each event the client generates an event signature that includes any semantic information that can be represented compactly. For VNC, this includes all events, except for the actual pixel data of a screen update. For any information that cannot be compacted, the client stores the original data in a local cache, and adds the cache index to the event signature. The local cache is managed by the client using an LRP (Least Recently Predicted) policy. To apply an event, the event semantics are extracted from the predicted event signature. If the event includes a cache index, the relevant information (e.g., actual pixel data) is extracted from the cache. If there is a cache miss, resulting from an earlier cache eviction, then the predicted event is dropped.

4.4 Predictor interface

When designing the interface between the display client and the predictor, we wanted to minimize the performance overhead suffered by the client due to additional processing caused by the predictor. At the same time, we wanted to be able to readily plug in different predictors. For these reasons, in our design the predictor runs asynchronously and in parallel to the display client as a separate process connected with a pair of pipes. This allows the client to interface with the predictor using the same mechanisms used to interface with the server, and also to prioritize events from the server over events sent by the predictor. Since all prediction processing is done outside

Symbol	Meaning
User input	
T_{target}	Desired round-trip time
Per-event control variables	
n	Number of predicted events needed per event
m	Number of immediate predicted events per event
Estimates	
r	Smoothed screen event rate
r_{inst}	Estimated instantaneous rate of screen events
T_{RTT}	Estimated network round-trip time
Parameters	
Δ	Interval for event rate estimator (typical: 10 ms)
α	Gain for estimator of r (typical: 0.001)
β	Variation factor for r (typical: 0.5)

Figure 6: Symbols used in description of prediction application policy.

of the client, there is no performance penalty due to the overhead of the prediction engine itself, only for communicating with it. Finally, despite keeping the VNC Viewer code base single-threaded, this design allows us to straightforwardly make use of an unused processor core, if available.

4.5 Prediction application policy

While there are many possible ways to use predictions to improve the performance of a remote display system, we restrict ourselves here to implementations that only affect the client. Furthermore, we focus on two scenarios that are challenging for current remote display implementations:

- Network connections in which packets experience high and/or highly variable latency (e.g., connections over WANs).
- Network connections in which packets experience high and/or highly variable loss rates (e.g., connections over wireless networks).

In both VNC and RDP, network connections are implemented using TCP. This causes a common problem in both scenarios: the *message* latency the client experiences can be high, and/or highly variable—many screen or user events will take a long time to traverse from client to server, or vice versa. Each message can contain multiple events, and can span multiple packets at the network layer. Packet loss or delay can result in the delay

of multiple events. From the perspective of the client, however, the goal is to have user events arrive promptly at the server, and correlated screen events to be returned promptly from the server.

A speculative remote display client uses prediction to overcome the network-induced event latency (and latency variance)—to trade the possibility of screen artifacts for a lower effective event latency (and lower variance). In order to employ a predictor that consumes user and screen events and can produce any number of predicted screen events, a speculative remote display client needs to answer several questions:

- 1 How many predicted screen events to ask for?
- 2 When to apply the predicted events?

We refer to how these questions are answered as the *prediction application policy*.

We now describe the policy that we designed, implemented, and evaluated in the context of a purely client-based variant of the VNC system, using the symbols introduced in Figure 6.

The user determines the goal of the system, T_{target} , the target event round-trip time. T_{target} can be changed at any time. The system continually estimates the current event round-trip time, T_{RTT} . This can be done using a ping estimate or the TCP RTT estimate.

As screen events flow into the system, it computes an estimate of their rate, r . Note that the instantaneous screen event rate is highly dynamic and dependent on user activity. Events can also be very closely spaced, indeed, the spacing can effectively be zero since multiple events can be packed into one message. Furthermore, the mechanisms for resolving time that the operating system makes available can affect the measurement of the instantaneous rate. For these reasons, we introduce an interval over which we estimate the instantaneous rate, Δ . For every Δ second window, we count the number of screen events, k , that occur within it, and estimate the instantaneous event rate as $r_{inst} = k/\Delta$. In the work reported here, we use $\Delta = 10\text{ms}$.

We derive a smoothed event rate estimate from these instantaneous estimates using exponential averaging. This smoothed event rate estimate is our prediction of the rate of events in the near future. The smoothed event rate estimate is computed:

$$r = \alpha r_{inst} + (1 - \alpha) \text{Prev}(r)$$

The value of α that we use in our experiments is 0.001.

Based on our smoothed screen event rate r , we now compute how many events we must predict on each new event that arrives, in order to reach the target:

$$m = (T_{RTT} - T_{target})r$$

Note that this analysis assumes that a prediction can be made on each event arrival. However, it is allowable for

a predictor in our system not to do so, or to return fewer than m predictions.³

We can also consider the system in terms of phase and frequency. Our estimate r is the frequency at which the system is operating. If the normal phase of the system is T_{RTT} , then T_{target} is the phase offset (“advanced phase”) we seek to achieve. In order to do so, we need to have executed the next $m = (T_{RTT} - T_{target})r$ events at any point in time. Furthermore, we need to be able to continue to execute events at rate r , regardless of whether they are predicted or server-provided events.

It is important to note that in our system predictions are produced in response to events, including screen events. This implies that a variation in message (and thus event) latency results in variation in the number of available predictions at any point in time. A long delay in receiving a new screen event means a long delay in producing predictions for events subsequent to it. Such a delay can result in an “underrun” of available predictions. We address such underruns (and the inability to make all needed predictions in some situations as described earlier) by requesting additional predictions on each event. In particular, we request $n = (1 + \beta)m$ predictions on each event. In our studies, $\beta = 0.5$.

The first m predictions are immediately applied to achieve the desired T_{target} , while the remaining βm predictions are kept in reserve in case predictions cannot be produced at the target rate r in the near future.

4.6 Reconciling predictions

The system maintains two queues for predicted screen events. A validation queue to track applied predictions and to enable error correction, as well as a pending queue to prevent prediction underruns. The queues are kept totally ordered by a (locally generated) event sequence number.

The *validation queue* stores those predicted screen events that have already been drawn to the screen. The predicted event at the head of the queue is the next event expected from the server. If an event from the server arrives and matches the predicted event at the head of the validation queue, the predicted event is removed from the queue and discarded (the prediction was correct). If the actual event does not match the predicted event then the system enters the error correction state. We discuss this state and its handling in Section 4.7.

The *pending queue* stores predictions that have not yet been applied. Predicted events are drained from the pending queue at rate r , at which time the drained event is applied and placed in the validation queue.

³Our k th order Markov predictor can only make m predictions if it has previously seen the last k events followed by at least one sequence of an additional m events.

On every event that produces the $n = (1 + \beta)m$ predictions described in Section 4.5, m of them are immediately drawn and placed into the validation queue. The βm “extra” predictions are placed into the pending queue. Note that every new group of n predictions overlaps in $n - 1$ places with the previous group of predictions. Several cases result from pairs of such overlapping events:

- If the earlier event has already been validated, the later event is discarded (and not drawn).
- If the earlier event has already been applied, but has not yet been validated, then the later event is discarded (and not drawn).⁴
- If the earlier event has not yet been applied (it is in the pending queue), then we replace the earlier event with the more recent one.

4.7 Prediction error correction

When the predicted and drawn event at the head of the validation queue is invalidated by the arrival of an actual screen event from the server, we know that the screen being displayed to the user is incorrect and we must repair it. There are numerous ways to do such a repair. Our system currently implements a simple, out-of-band approach.

On an invalidation, we know that the head event is wrong, and all subsequent events in the validation queue are suspect. Furthermore, even if a subsequent event is in fact correctly predicted, it can depend on the now incorrect screen state produced by the head event, and thus have resulted in an incorrect update of the screen. We thus consider all of the subsequent predicted events to be incorrect.

Predicted events in the pending queue are simply discarded. Nothing more needs to be done, as these events had not yet been drawn to the screen.

We compute a bounding rectangle over all of the events in the validation queue and request an update for that rectangle from the server. The events in the validation queue are then discarded. Although we do not implement it, it is clear that another approach would be to have a true undo log in the client that would permit entirely local rollback of the screen state.

The update we request from the server results in one or more screen events asynchronously being returned from

⁴The other possible choice would be to treat the earlier event as being invalidated by the later event, and thus prompt error correction. The problem with that approach is that the event with a given sequence number could be invalidated, and error corrected, up to m times—the new prediction could itself be invalidated, and so on. Additionally, although the newer prediction is more likely to be correct, it is not the case that the older prediction is now guaranteed to be incorrect. Given the cost of error correction, we decided to simply wait for validation by the actual screen event, which means that invalidation can occur at most 1 time per actual screen event.

the server. Currently, we do not handle these actual screen events in any special way. To do so would require that we modify the VNC protocol to indicate that the request is of a special type, and that the server response would include that type. For example, a “for repair” flag could be added to the request and the server could be modified to replicate this flag to any screen event it sends in response. In this paper, we focus on approaches to speculative remote display that require only client-side changes to existing systems. As we discuss in Section 6, we are exploring joint client/server speculative remote display protocols that enable this and several other optimizations.

4.8 User interface

In any speculative remote display system, the screen is allowed to become temporarily incorrect. Outside of having an oracle for a predictor, there is a tradeoff between the target RTT (T_{target}) goal and the degree of temporary screen artifacts introduced by bad predictions.

We have repeatedly found that user satisfaction with any specific systems software configuration or choice of parameters exhibits high variation, and that this variation can be exploited [5, 11]. Hence, we believe the tradeoff in this system should be made on an *individual* basis. Other researchers have also considered the effectiveness of putting the end user in direct control of systems-level decisions [21].

How do we let the individual user trade off between responsiveness and noise? In our system, we allow the user to set T_{target} directly, from 0 to T_{RTT} . The only extension to the VNC interface, is a simple gradient display representing the range of choices. The gradient floats above the VNC display on the right hand side of the screen. The display indicates the current choice by a bar and the current T_{target} number. The user can change the tradeoff at any time simply by clicking or dragging on the gradient.

5 User studies

We evaluated VNC/SRD by conducting two user studies.

5.1 Users

We conducted parallel user studies in the CS department at the University of Victoria (UV) and the EECS department at Northwestern University (NU). Our users consisted of six students at UV and seven students at NU, who were recruited via posters and email. As part of each study, the user rated his familiarity, on a 1–10 (10=most familiar) scale, with the OS and applications used in the study. Each user also rated his familiarity with remote

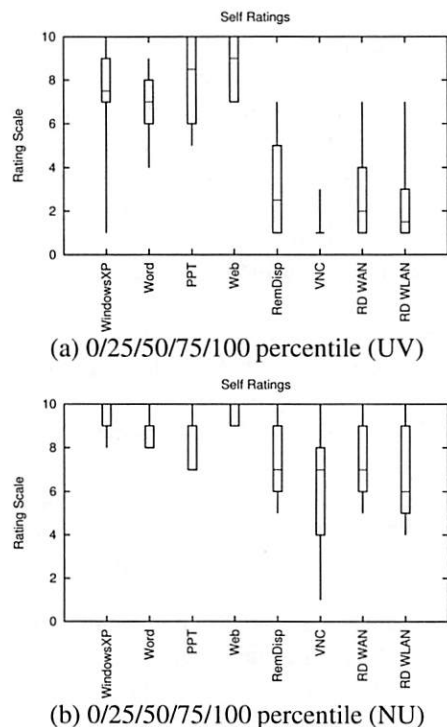


Figure 7: Self-ratings of users in studies.

display systems in general, their use in WAN and wireless environments, and with VNC specifically. This survey information summarizing the relevant background of our users is provided in Figure 7.⁵ Each user contributed approximately 90 minutes of his time. Users were not compensated.

5.2 Testbed

Both the UV and NU testbeds consisted of two machines (client and server) on a private network. At UV, each machine had an AMD Athlon 64 3500 series processor running at 2.2 GHz, 2 GB of RAM, and an NVIDIA 8800GTX graphics card running at 1280x1024 with 32 bit color. The monitors were 19 inch LCD displays. At NU, each machine had a 2.0 GHz Intel P4, 512 MB of RAM, and a Matrox G550 graphics card at 1600x1200 with 32 bit color driving a 20 inch LCD display. The user interacted solely with the client machine which ran Linux (Ubuntu “Gutsy Gibbon” version 7.10) and our VNC/SRD client. On the client machine, we used the netem extension of the iproute2 networking implementation in the Linux 2.6 kernel to allow controlled network conditions. The server machine ran Windows XP SP2 and an unmodified Real VNC server. Microsoft Office

⁵The specific survey instruments and protocol documents for the studies are available from <http://virtuoso.cs.northwestern.edu/vnc-srd-study>.

SRD configuration for User Studies

Cache size limit	100 MB
Markov state size limit	2000 states
Markov order k	2
Maximum number of predictions	64
α	0.001
β	0.5
Initial T_{target}	1/2 the actual RTT

Figure 8: Configuration settings for SRD client in User Studies.

Latency values for User Studies

50ms	Regional connections
100ms	National connections
300ms	International connections
10 ms with 10% loss	Bad wireless connection

Figure 9: Network scenarios.

2003 was installed on the server machine, as was Firefox 2.0.

Each user was presented with VNC/SRD running in full-screen, 24 bit mode, which, for the applications in our studies, is visually indistinguishable from using Windows XP natively. Superimposed at the lower right of the display was small grayscale slider described in Section 4.8. During the studies they were instructed to manipulate the slider as needed to make the system comfortable for them.

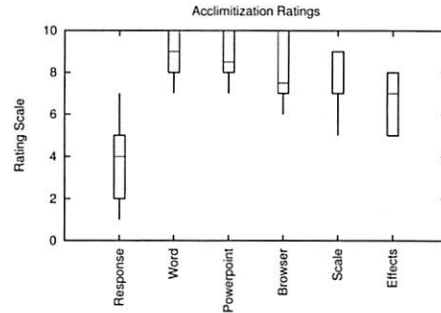
5.3 Parameters and scenarios

Figure 8 shows the parameters used to configure VNC/SRD in our studies. The cache size was set to 100 MB and the predictor limit was set to 2000 states. k was chosen based on the predictability study of Section 3, while α and β were chosen based on author experience with the system. Although it would be ideal to study the sensitivity of the choices for k , α , β and the memory limits of the cache and model, it's important to note that each individual data point in our work represents about two hours of proctor time.

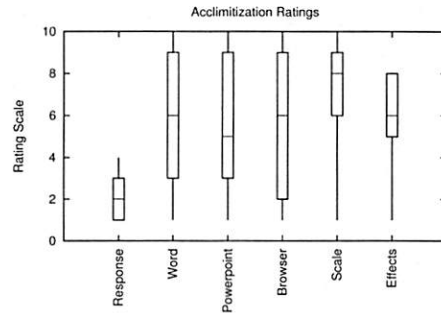
In our studies, we consider four different networking scenarios, as illustrated in Figure 9. Three of these scenarios represent increasing network latencies, while the fourth represents a wireless scenario with moderate loss. Note that as VNC communicates with TCP, the high loss rate scenario also represents one where the effective bandwidth is low.

5.4 Study design

A user in each of our studies performed the following tasks:



(a) 0/25/50/75/100 percentile (UV)



(b) 0/25/50/75/100 percentile (NU)

Figure 10: Results from acclimatization questionnaire in studies.

- Introduction (As long as needed). The user read a standardized one page description of the study, and description of the user interface.
- Self-rating (5 minutes). The user filled out a short survey rating familiarity with various software, and with remote desktop systems and VNC. (Figure 7 gives the results)
- Acclimatization (15 minutes). The user used the remote computer, particularly the three test applications, and became familiar with the VNC/SRD user interface. No network emulation was run at this point. The user then filled out a survey comparing the responsiveness of the computer to his most familiar one, his comfort with using the applications on the computer, and to what extent he understood the grayscale slider and its effects. This information is summarized in Figure 10.
- Word processing with Microsoft Word 2003. The user spent 15 minutes recreating a supplied document. Three network scenarios (R1,R2,R3) were chosen at random from among those in Figure 9 and given in a random order:
 - R1: The prediction model was cleared, the user interacted with the system for 5 minutes, and then answered three questions, given later.
 - R2: Same as R1, but with a different network scenario.
 - R3: Same as R1, but with a different network scenario.
- Presentation creation with Microsoft Powerpoint 2003. The user spent 15 minutes recreating a supplied

document with considerable drawing required. The same scenarios, questions, etc, as with Word were used.

- Web browsing using Firefox 2.0. For 15 minutes, the user visited a news web site (cnn.com), read an article, and then conducted web searches using Google on its topic in another window. The same scenarios, questions, etc, as with Word were used.

Notice that we evaluated only three of the four network scenarios with each individual user.

The original documents for the Word and Powerpoint tasks were supplied by us and were the same for all users. Word and Powerpoint ran in full screen mode. The users were not required to finish duplicating the document, only to get as far as they were comfortably able. For the web browsing task, the article and search windows were tiled side by side on the screen.

After each scenario-application pair, the user was asked to answer each of the following:

- Rate the statement "I was able to find a setting on the scale that was comfortable." (1 (Strongly Disagree) to 10 (Strongly Agree) (5=Neither))
- At the most comfortable setting I could find, the computer's responsiveness was (1 (Very bad) to 10 (Very Good) (5=Neither)).
- At the most comfortable setting I could find, the display errors were (1 (Unacceptable) to 10 (Unnoticeable) (5=Acceptable)).

5.5 Results

Figure 11 summarizes user responses to the three statements given in the preceding section. Figures 11(a)-(c) give the results for UV, while Figures 11(d)-(f) give the results for NU. The results are shown using Box plots where the whiskers correspond to minimum and maximum. The results shown in each individual figure are grouped first by the network scenario (which are arranged roughly in order of increasing latency) and second by the application. For example, Figure 11(a) summarizes UV user responses to "I was able to find a setting on the scale that was comfortable" (1 (Strongly Disagree) to 10 (Strongly Agree) (5=Neither)). The first three ranges in the graph correspond to user responses for the Word, Powerpoint, and Browsing tasks, all at 50 ms latency. The next three ranges are the same applications at 100 ms, and so on.

The data indicates that the majority of users tend to view the display artifacts as acceptable (Figures 11(c) and (f)). User perception of responsiveness, however, is highly variable (Figure 11(b) and (e)). For most scenario and application combinations, the majority of users registered agreement that to some extent that they were able to find comfortable settings.

5.6 Caveats

The primary issue with our studies, beyond their scale, is that there were not many instances where predictions were made, and thus the probability of prediction errors and screen artifacts was reduced. This is due to two factors. First, as we noted in Section 3.2, VNC screen event traffic exhibits only a small amount of predictability, much less than RDP screen event traffic. Second, our study design avoids ordering effects by, in part, creating a new model on each network/app scenario. However, this means that there are not as many past events on which predictions can be based, reducing the rate of predictions.

The users at NU were more familiar with remote display systems, and VNC in particular, than those at UV (Figure 7). The UV users generally found VNC/SRD's responsiveness to be closer to the computer they were most familiar with than did the NU users (Figure 10). This is likely due to the performance difference between the testbeds.

6 Design alternatives

We have so far described a design for a client-based speculative remote display system that allows a user to choose a desired message latency. As we described earlier, we use a Markov model-based predictor to generate predictions from repeated event sequences. However we note that there are many possible alternatives for incorporating event prediction into a display client.

Improving the client-based approach The most straightforward design alternative is to choose a different predictive model. VNC/SRD is designed in such a way that it is straightforward to integrate other online predictors. One approach would be to separately predict event types and parameters. For example, coordinates might be more readily, and cheaply, predicted using linear time series models. In separating types and parameters, it is not clear what the definition of an incorrect prediction would be. For instance, if an event's action was correctly predicted but its location was wrong, error correction could entail refetching all of the affected areas, or the predicted action could simply be moved to the correct location.

Another design option is further pre- and post-processing of event data. For example, currently, event locations are absolute coordinates, so two events are considered different if their locations differ by a single pixel, even if their actions are identical. Using differencing would be one method of handling this discrepancy.

There are also many alternatives in regards to the prediction application policy. Our current design tracks the

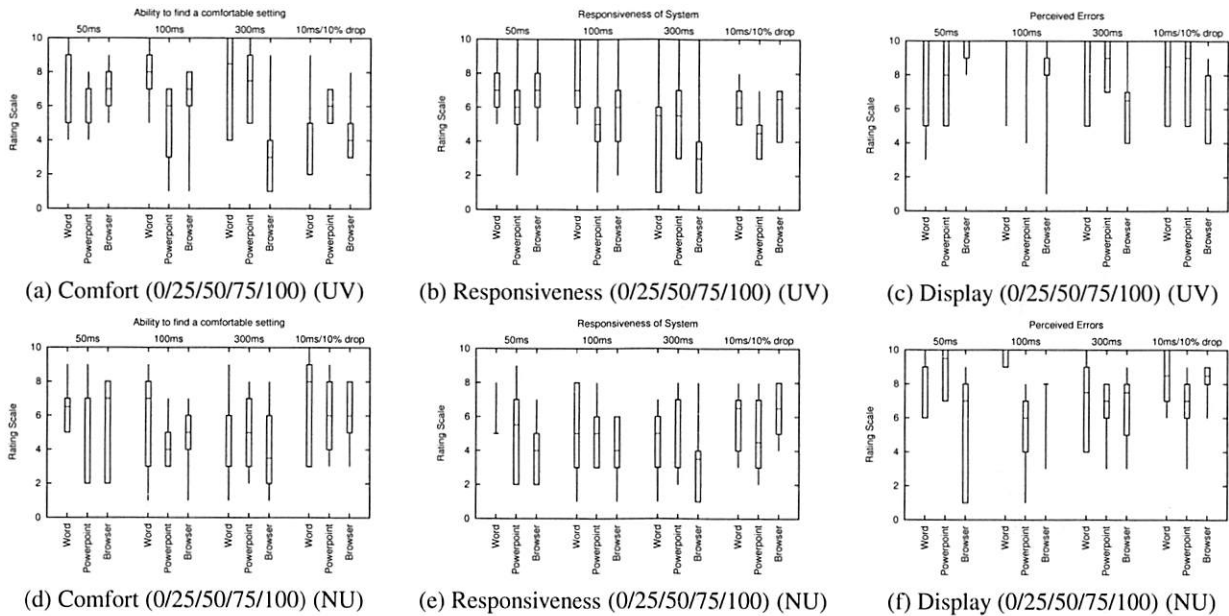


Figure 11: User study response analysis for survey questions described in Section 5.4

event arrival times but does not use them in the predictor or the policy decisions. A design that incorporated this information could use the Markov model to predict specific events and then use an estimator of some kind to determine at what time it should apply each of the predicted events. Such a scenario would allow a user to specify more precisely how far into the future they want the system to predict.

Currently our system uses a very simplistic mechanism for error recovery that involves retransmitting all the regions that were changed before an error was detected. We already mentioned that a simple undo-log to handle rollback would further improve performance, however there are other aspects of this problem to consider. We currently measure the user's tolerance of errors to control the aggressiveness of our prediction policy. However, this measurement could also be used to control an error correction policy. For instance, we could determine the error threshold that a user is willing to tolerate and enter error correction only when this threshold is exceeded. We could also selectively correct errors instead of correcting everything at once. For instance we could prioritize errors based on the number of pixels and the screen location.

Exploiting semantically richer protocols As we discussed in Section 3, the screen event predictability of the semantically richer RDP protocol is much higher than that of the simple VNC protocol, although the rate of events in RDP is also much higher. This raises the possibility that speculative remote display is more suitable

for semantically richer protocols, be they RDP or X11. If it is the case that semantically richer protocols benefit more, perhaps very high-level interactions, such as in AJAX web applications, are suitable for speculation.

Server-side cooperation and protocol extensions

With server-side cooperation, and necessary protocol extensions, a speculative remote display system could reduce the bandwidth demands of a remote display architecture. In such a system the client would make and apply predictions and send back the event signatures to the server. Upon receiving the predicted signatures the server could determine the correctness of the prediction and send either a revocation containing the signature or a full correction to the client. This would allow the server to avoid sending screen updates at all when a correct prediction was made, and would also allow the system to quickly detect and correct errors.

7 Conclusions

This paper introduced the idea of client-based speculative remote display, in which the client predicts and speculatively executes screen update events in order to ameliorate the high latencies and high loss rates seen on wide area and wireless networks, respectively. To evaluate the concept, we studied the predictability of Windows Remote Desktop (RDP) and VNC traffic, finding predictability in both, although much more in RDP than VNC. We developed a prototype speculative remote display client for VNC. VNC/SRD works without server

or protocol modifications, and gives the user an interface through which he can adjust the tradeoff between the aggressiveness of predictions (and thus effective reduction in latency) and the incidence of display artifacts. Two user studies were conducted to measure the usability of the system. The predictability studies, analysis of VNC/SRD's design, and the usability studies, argue for the feasibility of the speculative remote display concept, but raise a range of new questions regarding the structure of a speculative remote display system and protocol that we plan to address in the future.

References

- [1] BARRATTO, R., KIM, L., AND NIEH, J. Thinc: A virtual display architecture for thin-client computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)* (October 2005).
- [2] CHAPMAN, M. rdesktop: A remote desktop protocol client for accessing windows NT terminal server. <http://www.rdesktop.org>.
- [3] CLEARY, J. G., AND WITTEN, I. H. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications* 32, 4 (1984), 396–402.
- [4] DAVISON, B. D., AND HIRSH, H. Probabilistic on-line action prediction. In *Proceedings of the 1998 AAAI Spring Symposium on Intelligent Environments* (1998).
- [5] DINDA, P., MEMIK, G., DICK, R., LIN, B., MALLIK, A., GUPTA, A., AND ROSOFF, S. The user in experimental computer systems research. In *Proceedings of the Workshop on Experimental Computer Science (EXPCS)* (June 2007).
- [6] JEFFERSON, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 3 (1985), 404–425.
- [7] KARAM, M., AND TOBAGI, F. A. Analysis of the delay and jitter of voice traffic over the internet. In *Proceedings of IEEE INFOCOM* (2001), pp. 824–833.
- [8] KROEGER, T. M., AND LONG, D. D. E. Predicting file system actions from prior events. In *Proceedings of the USENIX Annual Technical Conference* (1996).
- [9] LAI, A., AND NIEH, J. Limits of wide-area thin-client computing. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2002).
- [10] LI, S., AND CHEN, C. Interest scheme: A new method for path prediction. In *Proceedings of the 5th ACM SIGCOMM workshop on Network and system support for games* (2006).
- [11] MALLIK, A., COSGROVE, J., DICK, R., MEMIK, G., AND DINDA, P. Pictel: Measuring user-perceived performance to control dynamic frequency scaling. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2008).
- [12] MICROSOFT. Remote desktop protocol (rdp) features and performance. Tech. rep., 2000.
- [13] NIGHTINGALE, E., CHEN, P., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)* (October 2005).
- [14] PADMANABHAN, V. N., AND MOGUL, J. C. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review* 27, 3 (1996), 22–36.
- [15] PANG, J., UYEDA, F., AND LORCH, J. R. Scaling peer-to-peer games in low bandwidth environments. In *Proceedings of the 6th International Workshop on Peer-to-Peer Systems* (2007).
- [16] PAXSON, V. End-to-end routing behavior in the Internet. In *Proceedings of the ACM SIGCOMM* (New York, August 1996), ACM Press, pp. 25–38.
- [17] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing* 2, 1 (January/February 1998).
- [18] SCHEIFLER, R. W., AND GETTYS, J. The x window system. *ACM Transactions on Graphics* 5, 2 (April 1986).
- [19] SCHMIDT, B., LAM, M., AND NORTH CUTT, J. The interactive performance of slim: A stateless thin client architecture. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)* (December 1999), pp. 32–47.
- [20] SOU, H., KANAYA, I., AND SATO, K. Computer aided drawing system based on prediction of drawing action. In *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology* (2006).
- [21] SOUSA, J., BALAN, R., POLADIAN, V., GARLAN, D., AND SATYANARAYANAN, M. Giving users the steering wheel for guiding resource-adaptive systems. Tech. Rep. CMU-CS-05-198, Department of Computer Science, Carnegie Mellon University, December 2005.
- [22] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. Auto-Bash: Improving configuration management with operating system causality analysis. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles* (2007).
- [23] TAPUS, C., SMITH, J. D., AND HICKEY, J. Kernel level speculative dsm. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid* (2003).
- [24] ZILLES, C. Increasing interactivity by predicting user actions. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Wild and Crazy Ideas Session (WACI)* (October 2004).

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *login*., the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see <http://www.usenix.org/membership/specialdisc.html>

For more information about membership, conferences, or publications, see <http://www.usenix.org>.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at <http://www.sage.org>.

Thanks to USENIX & SAGE Corporate Supporters

USENIX Patrons

Google
Microsoft Research
NetApp

USENIX Benefactors

Hewlett-Packard
IBM
Linux Pro Magazine
VMware

USENIX & SAGE Partners

Ajava Systems, Inc.
DigiCert® SSL Certification

FOTO SEARCH Stock Footage
and Stock Photography

Raytheon
rTIN Aps
Splunk
Tellme Networks
Zenoss

USENIX Partners

Cambridge Computer Services,
Inc.
cPacket Networks
EAGLE Software, Inc.
GroundWork Open Source
Solutions

Hyperic
Infosys
Intel
Interhack
Oracle
Ripe NCC
Sendmail, Inc.
Sun Microsystems, Inc.
UUNET Technologies, Inc.

SAGE Partner MSB Associates

ISBN-13: 978-1-931971-59-1



90000



9 781931 971591